Query Optimization for Dynamic Imputation

José Cambronero*
MIT CSAIL
jcamsan@csail.mit.edu

John K. Feser*
MIT CSAIL
feser@csail.mit.edu

Micah J. Smith*
MIT LIDS
micahs@mit.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

Missing values are common in data analysis and present a usability challenge. Users are forced to pick between removing tuples with missing values or creating a cleaned version of their data by applying a relatively expensive imputation strategy. Our system, ImputeDB, incorporates imputation into a cost-based query optimizer, performing necessary imputations on-the-fly for each query. This allows users to immediately explore their data, while the system picks the optimal placement of imputation operations. We evaluate this approach on three real-world survey-based datasets. Our experiments show that our query plans execute between 10 and 200 times faster than first imputing the base tables. Furthermore, we show that the query results from on-the-fly imputation differ from the traditional base-table imputation approach by 0–24%, in one error measure. Finally, we show that while dropping tuples with missing values that fail query constraints discards 3–88% of the data, on-the-fly imputation loses only 0–10%.

1. INTRODUCTION

Many databases have large numbers of missing or NULL values; these can arise for a variety of reasons, including missing source data (e.g., unknown facts), missing columns during data integration, de-normalized databases, or as a result of outlier detection and cleaning where NULL values are substituted for bad values [14]. Such NULL values can lead to incorrect or ill-defined query results [22], and as such removing these values from data before processing is often desirable.

One common approach is to manually replace missing values using a statistical or predictive model based on other values in the table and record. This process is called *imputation*. In this paper, we introduce ImputeDB, a new system designed to selectively apply imputation to a subset of records, on-the-fly, during query execution. *The key insight behind ImputeDB is that imputation only needs to be performed on the data relevant to a particular query and that this subset is generally much smaller than the entire database.* While traditional imputation methods work over the entire data set and replace all missing values, running a sophisticated imputation algorithm over a large data set can be very expensive—our experiments show that even a relatively simple decision tree algorithm takes just under 6 hours to train and run on a 600K row database.

A simpler approach might drop all rows with missing values, which can not only introduce bias into the results, but also result in discarding all of the data. In contrast to existing systems [3, 1], ImputeDB avoids imputing over the entire data set. Specifically, ImputeDB carefully plans the placement of imputation/row-drop operations inside the query plan, resulting in a significant speedup in query execution while reducing the number of dropped rows. Unlike previous work, the focus of our work is not on the imputation algorithms themselves (we can employ almost any such algorithm), but rather on placing imputation operations optimally in query plans. Specifically, our optimization algorithms generate the Pareto-optimal frontier of plans that trade imputation cost for result quality, and allow the analyst to specify their desired plan from this frontier.

Our approach enables an exploratory analysis workflow in which the analyst can issue standard SQL queries over a data set, even if that data has missing values. These queries execute between 10 and 200 times as fast as it takes to impute all missing values and then run queries (the traditional approach). Furthermore, the results obtained with this approach are similar in quality to those obtained with the traditional imputation approach — between 0 and 24 percent, in one error measure — in our empirical results over real-world data sets (see Section 4 for details).

ImputeDB is designed to enable early data exploration, by allowing analysts to run their queries without an explicit base-table imputation step.

1.1 Contributions

ImputeDB leverages a number of contributions to minimize imputation time while producing comparable results to traditional approaches. These contributions include:

- **Relational algebra extended with imputation:** We extend the standard relational algebra with two new operators to represent imputation operations: impute (μ) and drop (δ). The *Impute* operation fills in missing data values using any statistical imputation technique, such as chained-equation decision trees [3]. The *Drop* operation simply drops tuples which contain NULL values.
- **Model of imputation quality and cost:** We extend the traditional cost model for query plans to incorporate a measure of the *quality* of the imputations performed. We use the cost model to abstract over the imputation

*Author contributed equally to this paper.

Proceedings of the VLDB Endowment, Vol. 10, No. 7

```
SELECT income, AVG(white_blood_cell_ct)
FROM demo, exams, labs
WHERE gender = 2 AND
      weight >= 120 AND
      demo.id = exams.id AND
      exams.id = labs.id
GROUP BY demo.income
```

Figure 1: A typical public health query on CDC's NHANES data.

algorithm used. To add an imputation technique, it is sufficient to characterize it with two functions: one to describe its running time and one to describe the quality of its results.

- **Query planning with imputation:** We present the first query planning algorithm to jointly optimize for running time and the quality of the imputed results. It does so by maintaining multiple sets of Pareto optimal plans according to the cost model. By deferring selection of the final plan, we make it possible for the user to trade off running time and result quality.

2. MOTIVATING EXAMPLE

An epidemiologist at the CDC is tasked with an exploratory analysis of data collected from individuals across a battery of exams. In particular, she is interested in exploring the relationship between income and the immune system, controlling for factors such as weight and gender.

The epidemiologist is excited to get a quick and accurate view into the data. However, the data has been collected through CDC surveys (see Section 4.2), and there is a significant amount of missing data across all relevant attributes.

Before she can perform her queries, the epidemiologist must develop a strategy for handling missing values. She currently has two options: (1) she could drop records that have missing values in relevant fields, (2) she could use a traditional imputation package on her entire dataset. Both of these approaches have significant drawbacks. For the query pictured in Figure 1, (1) drops 1492 potentially relevant tuples, while from her experience, (2) has proven to take too long. The epidemiologist needs a more complete picture of the data, so (1) is insufficient, and for this quick exploratory analysis, (2) is infeasible.

She can run her queries immediately and finish her report if she uses ImputeDB, which allows her to write her query in standard SQL and automatically performs the imputations necessary to fill in the missing data. An example query is shown in Figure 1.

2.1 Planning with ImputeDB

The search space contains plans of varying performance and quality of results, as a product of the multiple possible locations for imputation. The user can influence the final plan selected by ImputeDB through a trade-off parameter $\alpha \in [0,1]$, where low α prioritizes the quality of the query results and high α prioritizes performance.

For the query in Figure 1, ImputeDB generates the plans in Figure 2 and Figure 3, when optimizing for quality and performance, respectively.

Figure 2 shows a quality-optimized plan that uses the impute operation μ, which employs a reference imputation strategy to fill in missing values rather than dropping tuples. It

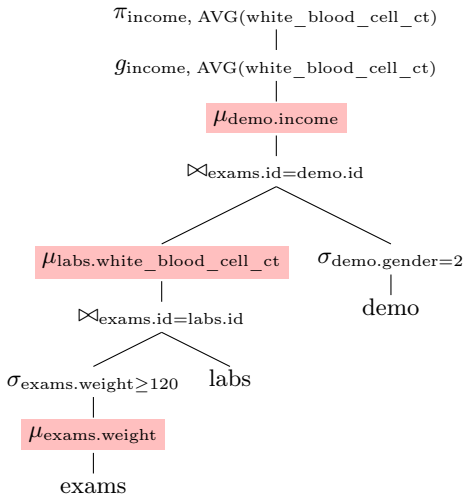

Figure 2: A quality-optimized plan for the query in Figure 1. The operators σ, π, $\bowtie$, and g are the standard relational selection, projection, join, and group-by/aggregate, μ and δ are specialized imputation operators (Section 3.1).

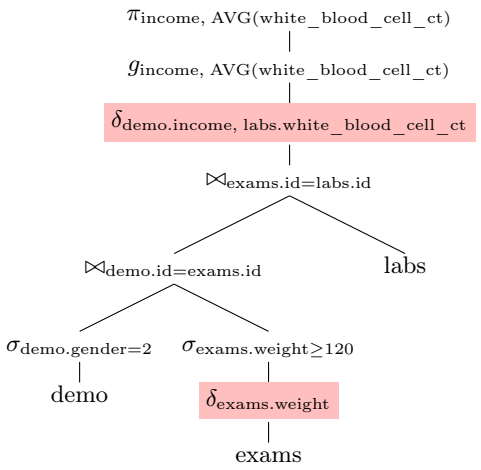

Figure 3: A performance-optimized plan for the query in Figure 1.The operators σ, π, $\bowtie$, and g are the standard relational selection, projection, join, and group-by/aggregate, μ and δ are specialized imputation operators (Section 3.1).

waits to impute `demo.income` until after the final join has taken place, though other imputations take place earlier on in the plan, some before filtering and join operations. Imputations are placed to maximize ImputeDB's estimate of the quality of the overall results. Meanwhile, Figure 3's performance-optimized plan uses the drop operation δ instead of filling in missing values. However, it is a significant improvement over dropping all tuples with missing data in any field, as it only drops tuples with missing values in attributes which are referenced by the rest of the plan. In both cases, only performing imputation on the necessary data yields a query that is much faster than performing imputation on the whole dataset and then executing the query.

2.2 ImputeDB Workflow

Now that the epidemiologist has ImputeDB, she can explore the dataset using SQL. She begins by choosing a value for α

and can adjust it up or down until she is satisfied with her query runtime. This iterative approach gives her immediate feedback.

Tens of queries later, our epidemiologist has a holistic view of the data and she has the information that she needs to construct a tailored imputation model for her queries of interest. Knowing that there may be a need to explore this data set further, she can easily incorporate her imputation model into ImputeDB for future use.

3. ALGORITHM

In order to correctly plan around missing values, we first extend the set of relational algebra operators (selection σ, projection π, join $\bowtie$, and group-by/aggregate g) with two new operators (Section 3.1). We then define the search space (Section 3.2) of plans to encompass non-trivial operations, such as joins and aggregations. Soundness of query execution is provided by our main design invariant, which guarantees traditional relational algebra operators must never observe a missing value in any attribute that they operate on directly (Section 3.3). The decision to place imputation operators is driven by our cost model (Section 3.4), which characterizes a plan based on estimates for the quality of the imputations performed and the runtime performance. Our imputation planning algorithm is agnostic to the type of imputation used (Section 3.6). Finally, we show that while our algorithm is exponential in the number of joins (Section 3.7), in practice, planning times are not a concern.

3.1 Imputation operators

We introduce two new relational operators to perform imputation: *Impute* (μ) and *Drop* (δ). Each operator takes arguments (C,R) where C is a set of attributes and R is a relation. *Impute* uses a machine learning algorithm to replace all NULL values with non-NULL values for attributes C in the relation R (discussed in detail in Section 3.6). In our experiments, we implement *Impute* using a chained-equation decision trees algorithm, but our system is agnostic to this choice. *Drop* simply removes from R all tuples which have a NULL value for some attribute in C. Both operators guarantee that the resulting relation will contain no NULL values for attributes in C.

3.2 Search space

To keep the query plan search space tractable, only plans that fit the following template are considered.

- All selections are pushed to the leaves of the query tree, immediately after scanning a table. We use a preprocessing step which conjoins together filters which operate on the same table, so we can assume that each table has at most one relevant filter.
- Joins are performed after filtering, and only left-deep plans are considered. This significantly reduces the plan space while still allowing plans with interesting join patterns.
- A grouping and aggregation operator is optional and if present will be performed after the final join.
- Projections are placed at the root of the query tree.

The space of query plans is similar to that considered in a canonical cost-based optimizer [2], with the addition of imputation operators appearing before and after traditional operators. Figure 4 shows a plan schematic for a query involving three tables, absent any imputation operators.

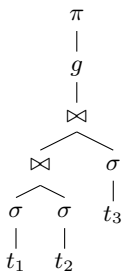

Figure 4: Query plan schematic for the type of traditional plans explored (absent imputation operators).

3.3 Imputation placement

We place imputation operators into the query plan so that no relational operator encounters a tuple containing NULL in an attribute that the operator examines, regardless of the state of the data in the base tables.

Imputation operators can be placed at any point, but there are cases where an imputation operator is required to meet the guarantee that no non-imputation operator sees a NULL value. To track these cases, we associate each query plan q with a set of dirty attributes $\textsc{Dirty}(q)$. An attribute c is *dirty* in some relation if the values for c contain NULL. We compute a dirty set for each table using the table histograms, which track the number of NULL values in each column. The dirty set for a query plan can be computed recursively as follows:

$$\textsc{Dirty}(q) = \begin{cases} \textsc{Dirty}(q') \setminus C & q = \delta_C(q') \text{ or } q = \mu_C(q') \\ \textsc{Dirty}(q') \cap C & q = \pi_C(q') \\ \textsc{Dirty}(q_l) \cup \textsc{Dirty}(q_r) & q = q_l \bowtie_\psi q_r \\ \textsc{Dirty}(q') & q = \sigma_\phi(q') \\ \textsc{Dirty}(t) & q = \text{some table } t \end{cases}$$

Note that Dirty over-approximates the set of attributes that contain NULL. For example, a filter might remove all tuples which contain NULL, but the dirty set would be unchanged. We choose to over-approximate to ensure that all NULL values are explicitly imputed or dropped.

3.4 Cost model

The cost of a plan q is expressed as a tuple $\langle \textsc{Loss}(q), \textsc{Time}(q) \rangle$. $\textsc{Loss}(q) \in [0,1]$ is an estimate of the amount of information lost by the imputation procedures used. $\textsc{Time}(q) \in (0,\infty)$ is an estimate of the runtime of a query q derived from table statistics and selectivity estimation of the query predicates.

3.4.1 Pareto-optimal plans

Given a query, ImputeDB produces a final set of plans P by only keeping plans that are not dominated by others in the search space. So for a search space S,

$$P = \{p \mid \nexists p' \in S.\ p \neq p' \wedge p' \succ p\}.$$

Definition 1. *We say that a plan p with cost $\langle l,t \rangle$ dominates a plan p' with cost $\langle l',t' \rangle$ if* $\textsc{Dirty}(p) = \textsc{Dirty}(p') \wedge ((l \leq l' \wedge t < t') \vee (l < l' \wedge t \leq t'))$. *We denote this as $p \succ p'$.*

This set P will be the Pareto frontier of S [19], and it contains the best option for all possible trade-offs of Time and Loss for the current query.

In order to pick a final plan from the frontier, our model introduces a trade-off parameter α, which is an upper bound

on the Loss value that the user is willing to tolerate relative to the minimum possible amongst frontier plans.

DEFINITION 2. *Let P be a set of plans. For $p \in P$, we say p is α-bound, if $(\text{Loss}(p) - \min_{p' \in P}(\text{Loss}(p')) \leq \alpha$. We denote this as p^α. We define the set of α-bound plans in P as P^α. We say a plan p^α is α-bound optimal if $\forall p' \in P^\alpha . \text{Time}(p) \leq \text{Time}(p')$.*

Given α and P, ImputeDB returns an α-bound optimal plan in P.

So in essence, α is a tunable parameter that determines whether the optimizer focuses on quality or on performance, as plans that lose a lot of information through dropping are also the fastest.

If $\alpha = 0.0$, then the optimal query should lose as little information as possible (at the expense of performance). If $\alpha = 1.0$, then the optimal query should have the fastest runtime (at the expense of quality).

3.4.2 Cardinality estimation

The computation of $\text{Time}(q)$ and $\text{Loss}(q)$ rely on having an accurate cardinality estimate of a query plan q and its sub-plans. These cardinality estimates are impacted not just by filtering or joining, as in the traditional relational calculus, but also by the imputation operators. For example, a drop operator will reduce the cardinality of the result while an impute operator will maintain the same cardinality as the input.

To compute cardinality, we maintain histograms for each column in the database. During query planning, each of the logical nodes in a query plan points to a set of histograms which describe the distribution of values in the output of the node. When the optimizer creates a new query plan, it copies the histograms of the sub-plans and modifies them as necessary to account for the new operation in the plan. For example it will redistribute the count of tuples with NULL in an imputed field. Algorithm 3 describes the process of generating new histograms from sub-plans.

3.4.3 Imputation quality

We use these histogram estimates to compute $\text{Loss}(Q)$, a measure averaged over imputation operations, as follows.

$$\text{L}(q) = \begin{cases} 1 + \text{L}(q') & q = \delta_C(q') \\ \frac{1}{\sqrt{\text{Attr}(q') * \text{Card}(q')}} + \text{L}(q') & q = \mu_C(q') \\ \text{L}(q'_l) + \text{L}(q'_r) & q = q'_l \bowtie_\psi q'_r \\ \text{L}(q') & q = \sigma_\phi(q'), q = \pi_C(q') \\ 0 & q = \text{some table } t \end{cases}$$

$$\text{N}(q) = \begin{cases} 1 + \text{N}(q') & q = \delta_C(q') \\ 1 + \text{N}(q') & q = \mu_C(q') \\ \text{N}(q'_l) + \text{N}(q'_r) & q = q'_l \bowtie_\psi q'_r \\ \text{N}(q') & q = \sigma_\phi(q'), q = \pi_C(q') \\ 0 & q = \text{some table } t \end{cases}$$

$$\text{Loss}(q) = \frac{\text{L}(q)}{\text{N}(q)}$$

Each imputation operator incurs a heuristic loss penalty. We define Loss as the average over the individual penalties incurred by μ, δ operators. Our recursive definition of L accumulates these penalties, and N counts the number of imputation operations. For example, δ contributes a loss of 1 (i.e. we lose all information that dropped tuples might have encoded). Meanwhile, imputing NULL fields with μ incurs a loss penalty $p \in (0,1]$. In this case, the penalty is inversely proportional to the square root of the number of cells available to train the imputation algorithm, which is defined as the product of the number of attributes and the number of tuples. Intuitively, imputation accuracy *increases* when more complete attributes and complete tuples are available; correspondingly, the information lost *decreases* with more values to work with. We encode the fact that the number of training observations has diminishing returns for imputation quality, and find that scaling the cell count by applying square root works well in practice.

3.4.4 Query runtime

To compute $\text{Time}(Q)$, an additive measure of runtime cost, we retain a simple set of heuristics used by a standard database.

Scan costs are estimated based on the number of tuples, page size, and a unit-less I/O cost per page. Join costs are estimated as a function of the two relations joined, any blocking operations, and repeated fetching of tuples (if required). Filtering, projecting, and aggregating are all done in memory and have low computational overhead so we assume negligible time costs for those operations.

We extend these heuristics to account for the new operators: *Drop* and *Impute*. *Drop* is a special case of a sequential scan, and its time complexity can be expressed as a function of the number of heap pages read and the I/O cost per page. The time computation for *Impute* depends on the properties of the underlying algorithm. ImputeDB treats the actual machine learning algorithm as a black box and simply queries it for a time estimate. In the implementation of our system, we use an iterative algorithm (see Section 3.6), where the runtime cost is a function of the number of attributes with no missing data, number of attributes with missing data being imputed, number of tuples, and the number of iterations for the algorithm.

In general, evaluating the time complexity for different *Impute* operators is complicated, as the properties of different algorithms can vary significantly, the underlying I/O cost is not easily extracted (especially in the case that the entire set of tuples does not fit in memory), and the CPU cost dominates (in contrast to the other operators, which do not consider CPU cost explicitly). For example, [18] finds that the time complexity of the *build tree* algorithm for one commonly-used class of decision trees is a function of the number of classes, several overhead constants, the parameterization of the partition and heuristic functions, and the *arity* (the number of subsets considered for each split). The *build tree* phase often does not even dominate computation, as post-processing steps like pruning, which are vital in achieving good performance, can have cost exponential in the height of the tree. Nonetheless, in practice, we find that a simple parameterization can be acceptable (i.e. yields intelligent query plans).

3.5 Query planning

The input to our query planner is a tuple (T,F,J,P,G,A):

- A set of tables T.
- A relation $F : T \times \Phi$ between tables and filter predicates.
- A relation $J : T \times \Psi \times T$ between tables and join predicates.
- A set of projection attributes P.
- An optional set of grouping attributes G and aggregator function A.

$$[\![Q[T]]\!] = \begin{cases} P & (T,P) \in Q \\ \varnothing & \text{otherwise} \end{cases}$$

$$[\![Q[T] \lhd p]\!] = Q \cup \{(T, P')\}$$

where

$$P' = \left\{ \begin{array}{r} q \in [\![Q[T]]\!] \cup \{p\}, \\ q \mid \textsc{Dirty}(q) = \textsc{Dirty}(p), \\ \forall q' \in [\![Q[T]]\!] . q \succ q' \end{array} \right\}$$

Figure 5: Semantics of a plan cache Q, where T is a set of tables and p is a query plan

The query planner must select a join ordering in addition to placing imputation operators as described in Section 3.3.

3.5.1 Plan cache

Before describing the operation of the query planner, we describe the semantics of a specialized data structure for holding sub-plans, called a *plan cache*. At a high level, a plan cache is a mapping from a set of tables to a set of dirty set-specific dominating plans that perform the required joins and filters over these tables. We store multiple plans rather than a single plan per dirty set, because maintaining a Pareto frontier as a plan progresses through the optimizer's pipeline allows us to obtain the final set of plans that best trade-off computation cost and imputation quality. Details of the plan-cache semantics are shown in Figure 5.

The plan cache uses the partial order of plans defined by $\langle \textsc{Dirty}, \textsc{Loss}, \textsc{Time} \rangle$ to collect sound and complete Pareto frontiers. For plans with the same dirty set, we can treat $\langle \textsc{Loss}, \textsc{Time} \rangle$ as a total order, allowing us to appropriately select the frontier. Plans with different DIRTY sets cannot be compared. The final Pareto frontier produced by the optimizer corresponds to the collection of plans associated with the empty dirty set.

3.5.2 Planning algorithm

The planner (Algorithm 2, with helper functions in Algorithm 1) operates as follows. First, it collects the set of attributes which are used by the operators in the plan or which are visible in the output of the plan (Line 2–Line 3). This set will be used to determine which attributes are imputed. Then, it constructs a plan cache. For each table used in the query, any available selections are pushed to the leaves and a search over imputation operations produces various possible plans, which are added to the cache (Line 7). If no selections are available, a simple scan is added to the plan cache (Line 9). Join order is jointly optimized with imputation placements and a set of plans encompassing all necessary tables is produced (Line 10). This set is now extended for any necessary grouping and aggregation operations (Line 12). After a similar step has been taken for projections, the planner now contains the plan frontier (i.e. the best possible set from which to pick our final plan). The final step in the planner is to find all plans that are α-bound and return the one that has the lowest runtime: the α-bound optimal plan.

The join order and imputation optimization is based on the algorithm used in System R [2], but rather than select the best plan at every point, we use our plan cache, which keeps track of the tables joined and the Pareto frontiers. The output of the join optimizer is the set of plans which satisfy all join predicates and lie along their respective frontier.

3.6 Imputation Strategies

ImputeDB is designed so that any imputation strategy can be plugged in with minimal effort and without changes to the optimizer, simply by adding time and loss functions (Section 3.4) which describe the new algorithm. For our purposes, an imputation strategy is a function $I(R,C) \rightarrow R'$ which fills in the missing attributes C in a relation R, returning a new relation R'. The flexibility of this approach aligns with the spirit of query optimization, which aims to reconcile performance with a declarative interface to data manipulation.

In principle, new imputation algorithms can be added, possibly targeted to a specific domain. However, ImputeDB's planner will consider an arbitrary subset of the attributes in any imputation, thus it is feasible to customize the *type* of algorithm but not necessarily a specific modelling relationships between attributes.

As a reference implementation, ImputeDB uses a general-purpose imputation strategy based on chained-equation decision trees. Chained-equation imputation methods [4] (sometimes called *iterative regression* [11]) impute multiple missing attributes being imputed by iteratively fitting a predictive model of one missing attribute, conditional on all complete attributes and all other missing attributes. Chained-equation decision trees algorithms are found to be effective [1] and are widely used in epidemiological domains [3]. For our experiments, we use Weka's `REPTree` implementation [24].

The reference algorithm used by the *Impute* operator proceeds by iteratively fitting decision trees for the target imputation attribute on a subset of the data. In each iteration, the missing values of a separate attribute are replaced with newly imputed values. In one *epoch* of the algorithm, each of the target imputation attributes is filled in with imputed values in a single iteration. With each epoch, the quality of the imputation improves as values progressively reflect more accurate relationships between attributes. The algorithm terminates when convergence is achieved (i.e. the imputed values do not change across epochs) or a fixed number of epochs is reached.

These imputation algorithms are commonly used as part of a larger *multiple imputation* process, in which multiple distinct copies of the complete data are generated and estimators are computed by averaging over the multiple datasets to form a single completed dataset [11]. In ImputeDB, we consider only single imputation, though multiple imputation could incorporated as another avenue of assessing imputation confidence.

3.7 Complexity

Our optimization algorithm builds off the approach taken by a canonical cost-based optimizer [2]. If imputation operators are ignored, we search the same plan space. Therefore, our algorithm has a complexity of at least $O(2^J)$, where J is the number of joins.

The addition of imputation operators increases the number of plans exponentially, as an imputation may be placed before any of the relational operators. We restrict imputations to two classes: those that impute only attributes used in the operator and those that impute all the attributes that are needed downstream in the query. By doing so we limit the number of imputations at any point to four: *Drop* or *Impute* over the attributes used in the operator or over all the downstream

Algorithm 1 Base algorithms for query planning with imputations.

Input: q is a query plan, C_l is a set of attributes that must be imputed in the output of this query plan, C_g is the set of attributes which are used in the final plan.
Output: A set of query plans Q with added imputation.

```
function AddImpute(q,C_l,C_g)
    D_must ← Dirty(q) ∩ C_l
    D_may ← D_must ∪ (Dirty(q) ∩ C_g)
    Q ← ∅
    if D_must = ∅ then
        Q ← Q ∪ {q}
    else
        Q ← Q ∪ {μ_{D_must}(q), δ_{D_must}(q)}
    if D_may ≠ ∅ then
        Q ← Q ∪ {μ_{D_may}(q), δ_{D_may}(q)}
    return Q
```

Input: t is a table, ϕ is a filter predicate, C_g is a set of attributes which are used in the final plan.
Output: A set of plans for filtering t with ϕ.

```
function OptFilter(t,φ,C_g)
    return {σ_φ(q) | q ∈ AddImpute(t, Attrs(φ), C_g)}
```

Input: q is a query plan, G is a set of grouping attributes, A is an aggregation function, C_g is a set of attributes which are used in the final plan.
Output: A set of plans for grouping and aggregating with G and A.

```
function OptGroupBy(q,G,A,C_g)
    C ← G ∪ Attrs(A)
    Q' ← AddImpute(q,C,C_g)
    return {GroupBy(q',G,A) | q' ∈ Q'}
```

Input: q is a query plan, P is a set of attributes to be projected at the top of the plan.
Output: A set of plans with the attributes in P projected.

```
function OptProject(q,P)
    return {π_P(q') | q' ∈ AddImpute(q,P,P)}
```

Input: A set of tables T, a plan cache Q containing at least one query plan for each table in T, and a relation $J : T \times \Psi \times T$ between tables and join predicates, C_g is the set of attributes which are used in the final plan.
Output: An updated plan cache Q' which contains at least one plan that combines all the tables in T.

```
function OptJoin(Q,T,J,C_g)
    for size ∈ 2...|T| do
        for S ∈ {all length size subsets of T} do
            for t ∈ S do
                S' ← S \ {t}
                for (t,ψ,t') ∈ J where t' ∈ S' do
                    C ← Attrs(ψ)
                    L ← {AddImpute(q,C,C_g) | q ∈ Q[S']}
                    R ← {AddImpute(q,C,C_g) | q ∈ Q[t]}
                    Q[S] ◁ {l ⋈_ψ r | l ∈ L, r ∈ R}
```

Algorithm 2 Top-level query planner with imputations.

Input: A set of tables T, a relation $F : T \times \Phi$ between tables and filter predicates, a relation $J : T \times \Psi \times T$ between tables and join predicates, a set of projection attributes P, an optional set of grouping attributes G and aggregator function A, and a parameter $\alpha \in [0,1]$ that expresses the trade-off between performance and imputation quality.
Output: An optimized query plan.

```
1:  function Plan(T,F,J,P,G,A)
2:      C_g ← ⋃_{φ∈F} Attr(φ)        ▷ Collect relevant attributes.
3:      C_g ← (⋃_{ψ∈J} Attr(ψ)) ∪ P ∪ G ∪ Attr(A) ∪ C_g
4:      Let Q be an empty plan cache.
5:      for t ∈ T do                  ▷ Add selections to the plan cache.
6:          if ∃φ : (t,φ) ∈ F then
7:              Q[{t}] ◁ OptFilter(Q,t,φ,C_g)
8:          else
9:              Q[{t}] ◁ {t}
10:     OptJoin(Q,T,J,C_g)            ▷ Optimize joins.
11:     B ← Q[T]                      ▷ Get the best plans for all tables.
12:     if G ≠ ∅ then                 ▷ Add optional group & aggregate.
13:         B ← ⋃_{q∈B} OptGroupBy(q,G,A,C_g)
14:     B ← ⋃_{q∈B} OptProject(q,P)   ▷ Add projections.
15:     return p ∈ B s.t. p is α-bound optimal.
```

Algorithm 3 An algorithm for in-plan histogram updates

Input: $\mathbb{H}$ is a map from attribute names to histograms, *op* is an operator node in a query plan
Output: Returns an updated map $\mathbb{H}'$

```
function UpdateHistograms(ℍ,op)
    if op = δ_C then
        ℍ' ← {c ↦ h' | h' = ℍ[c] without nulls if c ∈ C else ℍ[c]}
        {c ↦ h' | h' = ℍ'[c] scaled to min cardinality in ℍ'}
    else if op = μ_C then
        {c ↦ h' | h' = nulls redistributed by ℍ[c] if c ∈ C else ℍ[c]}
    else if op = σ then
        {c ↦ h' | h' = ℍ[c] scaled by Selectivity(σ)}
    else if op = ⋈_φ then
        {c ↦ h' | h' = ℍ[c] scaled to Cardinality(⋈_φ)}
    else
        ℍ
```

attributes. This modification increases the complexity of our planning algorithm to $O(2^{4J})$.

To motivate the restriction of imputation types, we consider the implications of allowing arbitrary imputations. If we allow any arbitrary subset of attributes to be imputed, then we would need to consider $O(2^{|D|+1})$ different imputation operators before each relational operator where D is the set of dirty attributes in all tables. This would increase the overall complexity of the algorithm drastically.

Finally, we note that for the queries we have examined, this exponential blowup does not affect the practical performance of our optimizer. Recall that the planner maintains different plans for different dirty sets, keeping only those plans that are not dominated by others. So in many cases we can drop some of the intermediate plans generated at each operator. The worst case complexity only occurs if the dirty sets tracked are

distinct through the entire planning phase. Planning times are discussed in Section 4.4.

4. EXPERIMENTS

For our experiments we plan and execute queries for three separate survey-based data sets, showing that our system is well suited for early dataset exploration. We show that ImputeDB performs an order of magnitude better than traditional base-table imputation and produces results of comparable quality.

4.1 Implementation

To evaluate the performance of our approach, we implemented a prototype system in Java, following a traditional iterator model. Our database handles a subset of standard SQL queries, including filters, joins, projection, grouping and aggregation. For simplicity, ImputeDB only handles tables containing integer data.

4.2 Data sets

We collect three data sets for our experiments. For all data sets, we selected a subset of the original attributes. We also transform all data values to an integer representation by enumerating strings and transforming floating-point values into an appropriate range.

4.2.1 CDC NHANES

For our first set of experiments, we use survey data collected by the U.S. Centers for Disease Control and Prevention (CDC). We experiment on a set of tables collected as part of the 2013–2014 National Health and Nutrition Examination Survey (NHANES), a series of studies conducted by the CDC on a national sample of several thousand individuals [5]. The data consists on survey responses, physical examinations, and laboratory results, amongst others.

There are 6 tables in the NHANES data set. We use three tables for our experiments:

- `demo`: demographic information of subjects
- `exams`: physical exam results
- `labs`: laboratory exam results

The original tables have a large number of attributes, in some cases providing more granular tests results or alternative metrics. We focus on a subset of the attributes for each table to simplify the presentation and exploration of queries. Table 1 shows the attributes selected, along with the percentage of NULL values for each attribute. For readability, we have replaced the NHANES attribute names with self-explanatory attribute names.

Attribute	% Missing
age_months	93.39
age_yrs	0.00
gender	0.00
id	0.00
income	1.31
is_citizen	0.04
marital_status	43.30
num_people_household	0.00
time_in_us	81.25
years_edu_children	72.45

(a) Demographics (`demo`). 10175 rows.

Attribute	% Missing
albumin	17.95
blood_lead	46.86
blood_selenium	46.86
cholesterol	22.31
creatine	72.59
hematocrit	12.93
id	0.00
triglyceride	67.94
vitamin_b12	45.83
white_blood_cell_ct	12.93

(b) Laboratory Results (`labs`). 9813 rows.

Attribute	% Missing
arm_circumference	5.22
blood_pressure_secs	3.11
blood_pressure_systolic	26.91
body_mass_index	7.72
cuff_size	23.14
head_circumference	97.67
height	7.60
id	0.00
waist_circumference	11.74
weight	0.92

(c) Physical Results (`exams`). 9813 rows.

Table 1: Missing value distribution for each table/attribute in CDC NHANES 2013–2014 data.

4.2.2 freeCodeCamp 2016 New Coder Survey

For our second set of experiments, we use data collected by freeCodeCamp, an open-source community for learning to code, as a part of a survey of new software developers [10]. The *2016 New Coder Survey* consists of responses by over 15,000 people to 48 different demographic and programming-related questions. The survey targeted users who were related to coding organizations.

We use a version of the data that has been pre-processed, but where missing values remain. For example, 46.6% of *commutetime* responses are missing. However, it is worth noting that some of the missing values are also expected, given the way the data has been de-normalized. For example, *bootcampploanyesno*, a binary attribute encoding whether a respondent had a loan for a bootcamp, is expected to be NULL for participants who did not attend a bootcamp.

We choose a subset of 17 attributes, which are shown in Table 2 along with the percentage of missing values.

4.2.3 American Community Survey

For our final experiment, we run a simple aggregate query over data from the American Community Survey (ACS), a comprehensive survey conducted by the U.S. Census Bureau. We use a cleaned version of the 2012 Public Use Microdata Sample (PUMS) data kindly provided by the authors of [1]. Given that the data had been cleaned, we artificially dirtied it by replacing 40% of the values uniformly at random with

Attribute	% Missing
age	12.85
attendedbootcamp	1.54
bootcampfinish	94.03
bootcampfulljobafter	95.93
bootcamploanyesno	94.02
bootcamppostsalary	97.89
childrennumber	83.65
citypopulation	12.74
commutetime	46.61
countrycitizen	12.59
gender	12.00
hourslearning	4.34
income	53.08
moneyforlearning	6.02
monthsprogramming	3.88
schooldegree	12.43
studentdebtowe	77.50

Table 2: Missing value distribution for each attribute in freeCodeCamp Survey Data (`fcc`)

NULL values. The final dataset consists of 671,153 rows and 37 integer columns.

4.3 Queries

We collect a set of queries (Table 3) that we think are interesting to plan. We believe that they could reasonably be written by a user in the course of data analysis.

The queries consist not only of projections and selections, but also interesting joins and aggregates. Our aim was to craft meaningful queries that would provide performance figures relevant to practitioners using similar datasets.

Our first set of queries is on the CDC data (Table 3a). Query 1 calculates the average cuff size for individuals based on their income data, with a constraint on height. Query 2 compares creatine levels for individuals with low, medium, and high incomes and above a certain weight. Query 3 extracts the average blood lead levels for children under 6 years of age. Query 4 calculates the average systolic blood pressure, by gender, for subjects with a body mass index indicating obesity. Query 5 calculates the average waist circumference for subjects above a certain height and weight.

Our second set of queries is on the freeCodeCamp data (Table 3b). Query 6 calculates the average income for higher-income survey participants, grouped by their bootcamp attendance. Query 7 estimates the average commute time of women from the United States who participated. Query 8 calculates the average amount of student debt based on school degree for survey participants who have student debt. Query 9 joins the freeCodeCamp data with data provided by the World Bank which summarizes GDP per-capita across various countries [23]. The query calculates the average GDP per-capita by grouping 18+ year old participants who attended bootcamp versus those who did not.

4.4 Results

For each query, we evaluate three different configurations: ImputeDB optimizing for quality ($\alpha = 0$), ImputeDB optimizing for performance ($\alpha = 1$), and imputing at the base tables followed by executing the query traditionally.

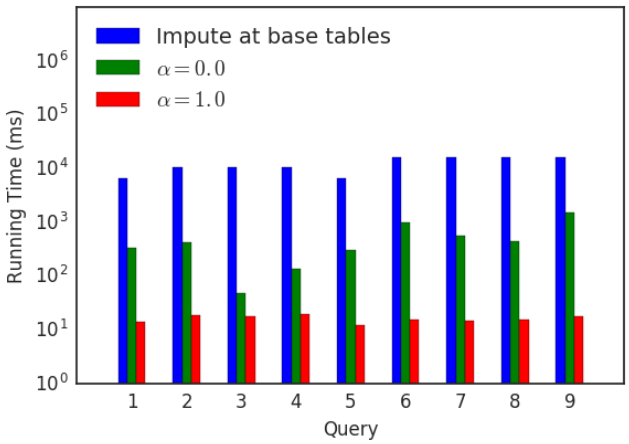

Figure 6: Query runtimes with imputation on base tables, ImputeDB optimizing for quality ($\alpha = 0$), and ImputeDB optimizing for performance ($\alpha = 1$). Quality-optimized, on-the-fly imputation provides an order of magnitude speedup over imputing on base tables. Efficiency-optimized on-the-fly imputation can provide another order-of-magnitude speedup at the expense of quality.

Figure 6 shows a summary of the performance results. The quality-optimized queries are an order-of-magnitude faster than imputing on the base tables. We can get another order-of-magnitude speedup when optimizing for performance. This performance differential means it is feasible to explore multiple imputations operations (including more expensive operators) when using ImputeDB, in contrast to the traditional approach of base table imputation.

In the median query across a variety of queries and choices of α, planning took 1.5 ms, constituting 1.8 percent of total runtime. In all cases, the optimizer returned a query plan within 34 ms, with times similar across values of α.

All experiments were run on a single Amazon Web Services EC2 `c4.xlarge` instance, with four 2.9 GHz Intel Xeon E5–2666 v3 virtual CPUs and 7.5 GiB of main memory, on Debian Linux. Each query with each value of α was run 200 times, after discarding a warm-up period.

4.4.1 *Accuracy vs. Base-table Imputation*

Table 4 shows the Symmetric-Mean-Absolute-Percentage-Error (SMAPE) [16] for ImputeDB's query results when compared to imputing on the base tables and executing the query on this cleaned base data. This measure indicates how much error is introduced by on-the-fly imputation compared to imputation on the base tables. In a real use-case, the data do not have ground-truth anyways, so the appropriate comparison is to this full-fledged impute on the base table.

For $\alpha = 0$ and $\alpha = 1$, we pair up tuples and compute SMAPE for all queries. That is, we compute tuple-wise absolute percentage deviations within each iteration of a query, and we report this value averaged over all iterations. We can see that optimizing for quality indeed reduces the SMAPE of query results. In general, the SMAPE relative to the base-imputation approach is low—between 0 and 24 percent—indicating that on-the-fly imputation produces similar results to imputation at the base tables. Notably, the results with largest error correspond most closely to one existing approach of dropping all NULLs.

#	Query
1	`SELECT income, AVG(cuff_size) FROM demo, exams WHERE demo.id = exams.id AND height >= 150 GROUP BY income;`
2	`SELECT income, AVG(creatine) FROM demo, exams, labs WHERE demo.id = exams.id AND exams.id = labs.id AND income >= 13 AND income <= 15 AND weight >= 63 GROUP BY income;`
3	`SELECT AVG(blood_lead) FROM demo, exams, labs WHERE demo.id = labs.id AND labs.id = exams.id AND age_yrs <= 6;`
4	`SELECT gender, AVG(blood_pressure_systolic) FROM demo, labs, exams WHERE demo.id = labs.id AND labs.id = exams.id AND body_mass_index >= 30 GROUP BY gender;`
5	`SELECT AVG(waist_circumference) FROM demo, exams WHERE demo.id = exams.id AND height >= 150 AND weight >= 100;`

(a) Queries on CDC data

#	Query
6	`SELECT attendedbootcamp, AVG(income) FROM fcc WHERE income >= 50000 GROUP BY attendedbootcamp;`
7	`SELECT AVG(commutetime) FROM fcc WHERE gender = "female" AND countrycitizen = "United␣States";`
8	`SELECT schooldegree, AVG(studentdebtowe) FROM fcc WHERE studentdebtowe > 0 AND schooldegree >= 0 GROUP BY schooldegree;`
9	`SELECT attendedbootcamp, AVG(gdp_per_capita) FROM fcc, gdp WHERE fcc.countrycitizen = gdp.country AND age >= 18 GROUP BY attendedbootcamp;`

(b) Queries on freeCodeCamp data

Table 3: Queries used in our experiments.

We also calculate the number of tuples used to produce each aggregate output. The count fraction columns in Table 4 show the number of tuples in the aggregate for $\alpha = 0$ and $\alpha = 1$ as a fraction of the number of tuples used when running the query on the imputed base table. This shows that when optimizing for performance, not quality, many tuples are dropped. Even in cases where the SMAPE reduction between $\alpha = 1$ and $\alpha = 0$ is small (Query 2, Query 6) the tuple count is significantly different. In these cases, the aggregate value is not significantly impacted by the missing data. In particular, if values are missing uniformly at random, the aggregate should not be affected. However, if the missing data is biased then the aggregate will have a significant error. This highlights a challenge for a user handling data imputation traditionally: it is unclear if the missing data will have a large or small negative impact on their analysis until they have paid the cost of running it. By using ImputeDB this cost can be lowered significantly.

4.4.2 *Runtime of Base-table Imputation*

In many real-world cases, applying the imputation step at the base table is prohibitively expensive. To illustrate the increasing difficulty of such an approach as datasets scale, we run the following query over the ACS dataset:

```
SELECT AVG(c0) FROM acs_dirty;
```

	SMAPE		Count Fraction	
Query	$\alpha=0$	$\alpha=1$	$\alpha=0$	$\alpha=1$
1	0.34	1.24	1.00	0.88
2	2.65	2.41	1.00	0.33
3	0.49	2.14	1.00	0.57
4	0.31	0.30	0.89	0.80
5	0.01	0.20	1.00	0.94
6	0.81	5.73	0.91	0.57
7	0.30	0.57	1.00	0.66
8	6.34	23.87	1.00	0.22
9	0.26	0.34	1.00	0.97

Table 4: Symmetric-Mean-Absolute-Percentage-Error for queries run under different α parameterizations. To calculate SMAPE, ImputeDB results are compared to query results returned from executions with base table imputation. Queries optimized for quality ($\alpha = 0$) generally achieve lower error than queries optimized for efficiency ($\alpha=1$). The count fraction column shows the number of tuples used in calculating each aggregate as a fraction of the number of tuples used when running the same query after imputing on the base table. A lower count share reflects more potential for errors.

Applying the imputation operation to the base table is extremely expensive, as it imputes over all attributes in all rows, in our case completing in 355 minutes. In contrast, ImputeDB executes a quality-optimized version in 4 seconds and a runtime-optimized version in 1 second. This highlights the potential benefit of using our system for early data exploration. Note that the performance differential relative to base table imputation would likely widen further if selection predicates were added to the query.

For every imputation inserted into the query plan, a new statistical model is instantiated and trained before being used for imputation. Although it is tempting to further optimize query execution in ImputeDB by pre-training imputation models while the analyst begins to make queries, this is not desirable because the specific rows of data, set of complete attributes, and set of attributes to impute are unknown until runtime.

5. RELATED WORK

There is related work in three primary areas: statistics, database research, and forecasting.

5.1 Missing Values and Statistics

Imputation of missing values is widely studied in the statistics and machine learning communities. As highlighted in [11], missing data can appear for a variety of reasons. It can be missing uniformly at random or conditioned on existing values (observed and missing). Methods in the statistical community focus on correctly modeling relationships between attributes to account for different forms of missingness. For example, [3] discuss the usage of iterative decision trees for imputing missing data.

[1] analyzes the performance of various multiple imputation techniques on the American Community Survey dataset. The computational difficulties of imputing on large base tables are well-known and can limit approaches. For example, [1] finds that one approach (MI-GLM) is prohibitively expensive when attempting to impute on data that includes variables with potentially large domains (ten categories in their case). In contrast, ImputeDB allows users to specify a trade-off between imputation quality and runtime performance, allowing users to perform queries directly on the entire dataset. Furthermore, the query planner's imputation is guided by the requirements of each specific query's operators, rather than requiring broad assumptions about query workloads.

5.2 Missing Values and Databases

There is a long history in the database community surrounding the treatment of NULL. As early as 1973, [6] provided a treatment of the semantics of NULL. Multiple papers have described various (at times conflicting) treatments of NULLs [12]. ImputeDB's main design invariant—no relational operator sees missing values for attributes it must operate on nor do users see missing data—eliminates the need to handle NULL value semantics, while guaranteeing query evaluation soundness.

Database system developers and others have worked on techniques to automatically detect dirty values, whether missing or otherwise, and rectify the errors if possible. A survey of methods and systems is provided in [13].

In [25], queries over databases with missing values are processed directly using a statistical approach, taking advantage of correlations between attributes. The tuples that match the original query as well as a ranking of tuples with incomplete data that may match are returned. Our work differs in that we allow any well-formed statistical technique to be used for imputation and focus on returning results to the analyst as if the database had been complete.

Designers of data stream processing systems frequently confront missing values and consider them carefully in query processing. Often, if sensor error is the cause of missing values, values can be imputed with high confidence. In [8], feedback punctuation is used to dynamically reconfigure the query plan for state-dependent optimizations as data continues to arrive. One of the applications of this framework is to avoid expensive imputations as real-time conditions change.

Other work has looked at integrating statistical models into databases. For example, BayesDB [17] provides users with a simple interface to leverage statistical inference techniques in a database. Non-experts can use a simple declarative language (an extension of SQL), to specify models which allow missing value imputation, amongst other broader functionality. Experts can further customize strategies and express domain knowledge to improve performance and accuracy.

While BayesDB can be used for value imputation, this step is not framed within the context of query planning, but rather as an explicit statistical inference step within the query language, using the `INFER` operation.

BayesDB provides a great alternative for bridging the gap between traditional databases and sophisticated modeling software. ImputeDB, in contrast, aims to remain squarely in the database realm, while allowing users to directly express queries on a potentially larger subset of their data.

ImputeDB's cost-based query planner is partially based on the seminal work developed for System R's query planning [2]. However, in contrast to System R, ImputeDB performs additional optimizations for imputing missing data and uses histogram transformations to account for the way relational and imputation operators affect the underlying tuple distributions.

5.3 Forecasting and Databases

[21] introduce the idea of incorporating time-series forecast operators into databases, along with the necessary relational algebra extensions. Their work explores the theoretical properties of forecast operators and generalizes them into a family of operators, distinguished by the type of predictions returned. They highlight the use of forecasting for replacing missing values. In their follow on work [20], they identify various equivalence and containment relationships when using forecast operators, which could be used to perform query plan transformations that guarantee the same result. They explore forecast-first and forecast-last plans, which perform forecasting operations before and after executing all traditional relational operators, respectively.

[9] describe the architecture of a DBMS with integrated forecasting operations for time-series, detailing the abstractions necessary to do so.

In contrast to this work, ImputeDB is targeted at generic value imputation, not necessarily tailored to time-series. The optimizer is not based on equivalence transformations, nor are there guarantees of equal results under different conditions. Instead, the optimizer allows users to pick their trade-off between runtime cost and imputation quality. The search space considered by our optimizer is broader, not just forecast-first/forecast-last plans, but rather imputation operators can be placed anywhere in the query plan (with some restrictions). The novelty of our contribution lies in the successful incorporation of imputation operations in non-trivial query plans with cost-based optimization.

[7] describes the Fa system and its declarative language for time-series forecasting. Their system automatically searches the space of attribute combinations/transformations and statistical models to produce forecasts within a given accuracy threshold. Accuracy estimates are determined using standard techniques, such as cross-validation.

Similarly to Fa, ImputeDB provides a declarative language, as a subset of standard SQL. ImputeDB, however, is not searching the space of possible imputation models, but rather the space of query plans that incorporate imputation operators. Another major point of distinction between ImputeDB and Fa is that the latter doesn't consider trade-offs between accuracy and computation time, but rather returns the most accurate forecast (with some stopping criterion).

6. CONCLUSION

Our evaluation of ImputeDB shows that missing values and their imputation can be successfully integrated into the relational calculus and existing cost-based plan optimization frameworks. We implement imputation actions, such as dropping or imputing values with a machine learning technique, as operators in the algebra and use a simple yet effective cost model to consider trade-offs in imputation quality and runtime. We show that different values for the trade-off parameter can yield substantially different plans, allowing the user to express their preferences for performance on a per-query basis. Our experiments with the CDC NHANES, freeCodeCamp and ACS datasets show that ImputeDB can be used successfully with real-world data to run standard SQL queries over data with missing values. We craft a series of realistic queries, meant to resemble the kind of questions an analyst might ask during the data exploration phase. The plans selected for each query execute an order-of-magnitude faster than the standard approach of imputing on the entire base tables and then formulating queries. Furthermore, the difference in query results between the two approaches is shown to be small in all queries considered (0-24% error, Section 4.4). Data analysts need not commit to a specific imputation strategy and can instead vary this across queries.

We discuss the long history of dealing with missing data both in the statistical learning and database communities. Similar to existing work, we consider the impact of NULL values in databases and develop a simple set of invariants to successfully plan around them. In contrast to existing statistical learning work, our emphasis is not on the specific algorithm used to impute but rather on the placement of imputation steps in the execution of a query. In contrast to existing database work, we incorporate imputation into a cost-based optimizer and hide any details regarding missing values inside the system, allowing users to use traditional SQL as if the database were complete. While prior work has incorporated operations such as prediction into their databases, this has been domain-specific in some cases, and in others, they have not integrated these operations into the planner. The novelty of our contribution lies in the formalization of missing value imputation for query planning, which results in performance gains over traditional approaches. This approach acknowledges that different users performing different queries on the same dataset will likely have varying imputation needs and that execution plans should appropriately reflect that variation.

6.1 Future Work

ImputeDB opens up multiple avenues for further work.

- **Imputation confidence:** Obtaining confidence measures for query results produced when missing values are imputed with a statistical model is another possible improvement to the system. These measures could be obtained in general using resampling methods like cross-validation and bootstrapping[15]. In contrast to standard confidence calculations, one added complexity is that imputation takes place at various points in the plan, so the composition of results through standard query operations needs to be taken into account when computing confidence measures. In addition, multiple imputation could be used within query execution and confidence measures like variances could similarly propagate through the query plan.

- **Imputation operators:** We explored a subset of possible imputation operators. In particular, we explored two variants of the *Drop* and *Impute* operations. The system could be extended to consider a broader family of operators. As the search space grows, there will likely be additional steps needed in order to avoid sub-optimal plans.

- **Adaptive query planning:** Currently, the imputation is local to a given query, meaning no information is shared across queries. An intriguing direction would be to take advantage of imputation and execution data generated by repeated queries. Note that this goes beyond simply caching imputation results, and instead could entail operations such as extending intermediate results with prior imputation values to improve accuracy, or pruning out query plans which have been shown to produce low-confidence imputations.

References

[1] O. Akande, F. Li, and J. Reiter. "An Empirical Comparison of Multiple Imputation Methods for Categorical Data". In: *arXiv preprint arXiv:1508.05918* (2015).

[2] M. W. Blasgen et al. "System R: An architectural overview". In: *IBM systems journal* 20.1 (1981), pp. 41–62.

[3] L. F. Burgette and J. P. Reiter. "Multiple imputation for missing data via sequential regression trees". In: *American Journal of Epidemiology* 172.9 (2010), pp. 1070–1076.

[4] S. van Buuren and K. Groothuis-Oudshoorn. "mice: Multivariate Imputation by Chained Equations in R". In: *Journal of Statistical Software* (2011).

[5] Center for Disease Control. *National Health and Nutrition Examination Survey (2013-2014)*. `https://wwwn.cdc.gov/nchs/nhanes/ContinuousNhanes/Default.aspx?BeginYear=2013`. Accessed: 2016-09-03.

[6] E. F. Codd. "Understanding relations". In: *ACM SIGMOD Record* 5.1 (1973), pp. 63–64.

[7] S. Duan and S. Babu. "Processing forecasting queries". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 711–722.

[8] R. Fernández-Moctezuma, K. Tufte, and J. Li. "Inter-Operator Feedback in Data Stream Management Systems via Punctuation". In: *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. 2009.

[9] U. Fischer et al. "Towards integrated data analytics: Time series forecasting in DBMS". In: *Datenbank-Spektrum* 13.1 (2013), pp. 45–53.

[10] FreeCodeCamp. *2016 New Coder Survey*. `https://www.kaggle.com/freecodecamp/2016-new-coder-survey-`. Accessed: 2016-09-03.

[11] A. Gelman and J. Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press, 2006.

[12] J. Grant. "Null values in a relational data base". In: *Information Processing Letters* 6.5 (1977), pp. 156–157.

[13] J. M. Hellerstein. "Quantitative data cleaning for large databases". In: *United Nations Economic Commission for Europe (UNECE)* (2008).

[14] W. Kim et al. "A Taxonomy of Dirty Data". In: *Data Mining and Knowledge Discovery* 7.1 (Jan. 1, 2003), pp. 81–99. ISSN: 1384-5810, 1573-756X. DOI: `10.1023/A:1021564703268`.

[15] R. Kohavi et al. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *IJCAI*. Vol. 14. 2. Stanford, CA. 1995, pp. 1137–1145.

[16] S. Makridakis and M. Hibon. "The M3-Competition: results, conclusions and implications". In: *International Journal of Forecasting* 16.4 (2000). The M3- Competition, pp. 451–476. ISSN: 0169-2070. DOI: `10.1016/S0169-2070(00)00057-1`. URL: `http://www.sciencedirect.com/science/article/pii/S0169207000000571`.

[17] V. Mansinghka et al. "BayesDB: A probabilistic programming system for querying the probable implications of data". In: *arXiv preprint arXiv:1512.05006* (2015).

[18] J. K. Martin and D. S. Hirschberg. *The time complexity of decision tree induction*. Citeseer, 1995.

[19] V. Pareto. *Cours d'économie politique*. Vol. 1. Librairie Droz, 1964.

[20] F. Parisi, A. Sliva, and V. Subrahmanian. "A temporal database forecasting algebra". In: *International Journal of Approximate Reasoning* 54.7 (2013), pp. 827–860.

[21] F. Parisi, A. Sliva, and V. Subrahmanian. "Embedding forecast operators in databases". In: *International Conference on Scalable Uncertainty Management*. Springer. 2011, pp. 373–386.

[22] D. B. Rubin. "Inference and Missing Data". In: *Biometrika* 63.3 (1976), pp. 581–592. ISSN: 0006-3444. DOI: `10.2307/2335739`. JSTOR: `2335739`.

[23] The World Bank. *World Development Indicators: GDP Per Capita*. `http://databank.worldbank.org/data/`. Accessed: 2016-02-12.

[24] I. H. Witten et al. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[25] G. Wolf et al. "Query processing over incomplete autonomous databases". In: *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, pp. 651–662.