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 140 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 0–8%. Finally, we show that while dropping tuples with missing values that fail query constraints discards 6–78% of the data, on-the-fly imputation loses only 0–21%.

1. INTRODUCTION

Many databases have large numbers of missing or NULL values; these can arise for a variety of reasons, including missing source data, missing columns during data integration, denormalized databases, or outlier detection and cleaning [15]. Such NULL values can lead to incorrect or ill-defined query results [23], 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 new methods and theory to selectively apply imputation to a subset of records, dynamically, during query execution. We instantiate these ideas in a new system, ImputeDB[1]. *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.

[*]Author contributed equally to this paper.
[1]`https://github.com/mitdbg/imputedb`

Proceedings of the VLDB Endowment, Vol. 10, No. 11
Copyright 2017 VLDB Endowment 2150-8097/17/08.

A simpler approach might drop all rows with any missing values, which can not only introduce bias into the results, but also result in discarding much of the data. In contrast to existing systems [2, 4], ImputeDB avoids imputing over the entire data set. Specifically, ImputeDB carefully plans the placement of imputation or 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 trade-offs between 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. Using dynamic imputation, these queries execute between 10 and 140 times faster than the traditional approach of first imputing all missing values and then running queries. Empirical results obtained using dynamic imputation on real-world datasets show errors within 0 to 8 percent of the traditional approach (see Section 5 for more details). Alternately, configuring dynamic imputation to prioritize query runtime yields further speedups of 450 to 1400 times, with errors of 0 to 20 percent.

1.1 Contributions

ImputeDB is designed to enable early data exploration, by allowing analysts to run their queries without an explicit base-table imputation step. To do so, it 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 [4]. The *Drop* operation simply drops tuples which contain NULL values.

```
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.

- **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 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 5.1), and there is a significant amount of missing data across all relevant attributes.

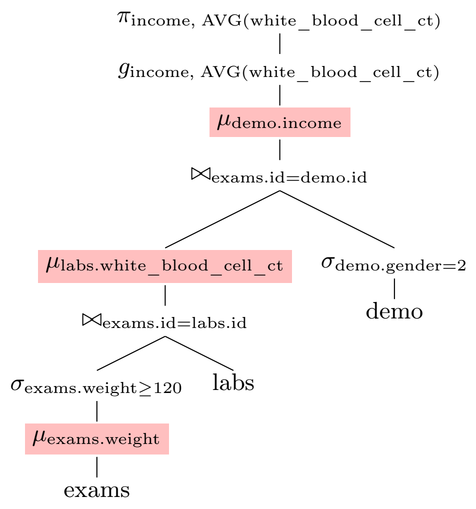
Before she can perform her queries, the epidemiologist must develop a strategy for handling missing values. She currently has two options: (1) she could drop records that have missing values in relevant fields, (2) she could use a traditional imputation package on her entire dataset. Both of these approaches 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 takes 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

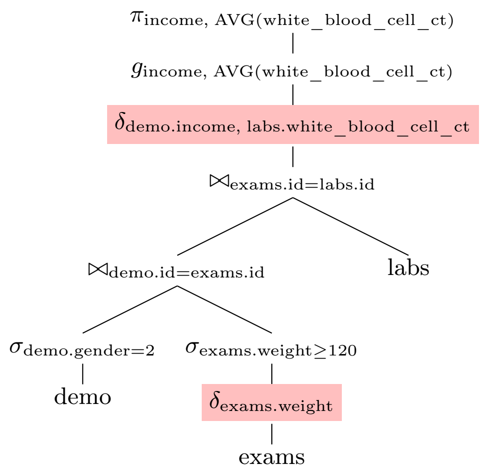
The search space contains plans with varying performance and imputation quality characteristics, as a product of the multiple possible locations for imputation. The user can influence the final plan selected by ImputeDB through a trade-off parameter $\alpha \in [0,1]$, where low α prioritizes the quality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

For the query in Figure 1, ImputeDB generates several plans on an optimal frontier. Figure 2a shows a quality-optimized plan that uses the *Impute* operator μ, which employs a reference imputation strategy to fill in missing values rather than



$\pi_{\text{income, AVG(white\_blood\_cell\_ct)}}$
$g_{\text{income, AVG(white\_blood\_cell\_ct)}}$
$\mu_{\text{demo.income}}$
$\bowtie_{\text{exams.id=demo.id}}$
$\mu_{\text{labs.white\_blood\_cell\_ct}}$
$\sigma_{\text{demo.gender=2}}$
demo
$\bowtie_{\text{exams.id=labs.id}}$
$\sigma_{\text{exams.weight}\geq 120}$
labs
$\mu_{\text{exams.weight}}$
exams

(a) A quality-optimized plan for the query in Figure 1.



$\pi_{\text{income, AVG(white\_blood\_cell\_ct)}}$
$g_{\text{income, AVG(white\_blood\_cell\_ct)}}$
$\delta_{\text{demo.income, labs.white\_blood\_cell\_ct}}$
$\bowtie_{\text{exams.id=labs.id}}$
$\bowtie_{\text{demo.id=exams.id}}$
labs
$\sigma_{\text{demo.gender=2}}$
$\sigma_{\text{exams.weight}\geq 120}$
demo
$\delta_{\text{exams.weight}}$
exams

(b) A performance-optimized plan for the query in Figure 1.

Figure 2: 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), and can be mixed throughout a plan. The imputation operators are highlighted.

dropping tuples. It 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. On the other hand, Figure 2b shows a performance-optimized plan that 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

Table 1: Notation used in Section 3.

Operator	Description
Attr(·)	Attributes used in a predicate.
Time(·)	Estimated time to run a plan.
Penalty(·)	Impute quality penalty of a plan.
Card(·)	Estimated cardinality of a plan.
$\widehat{Var}(\cdot)$	Estimated variance of a column.
Dirty(·)/Clean(·)	Dirty/clean attributes in a plan.
$Q[\cdot]$	Plan cache access operator.
$\cdot\lhd\cdot$	Plan cache insertion operator.

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

As an alternative strategy, the analyst can start by executing her query with $\alpha = 1$ and incrementally reducing α, as long as execution of the query completes acceptably quickly. ImputeDB's planning algorithm can also produce a set of plans with optimal time-quality trade-offs, which allows the user to adjust her choice of α to select from these plans.

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.

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 of the quality of the imputations performed and the runtime performance. Our planning algorithm is agnostic to the type of imputation used (Section 3.5). Finally, we show that while our algorithm is exponential in the number of joins (Section 3.7), planning times are not a concern in practice.

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 3.5). *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 pre-processing step which conjoins together filters which operate on the same table, so we can assume that each table has at most one relevant filter.

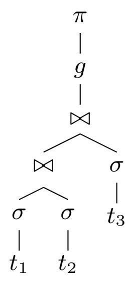


Figure 3: Query plan schematic for the type of traditional plans explored (absent imputation operators).

- 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 [3], with the addition of imputation operators. Figure 3 shows a plan schematic for a query involving three tables, absent any imputation operators. Plans produced by ImputeDB can mix both *Impute* and *Drop* 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 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 a tuple $\langle \textsc{Penalty}(q), \textsc{Time}(q) \rangle$. $\textsc{Penalty}(q) \in (0,1]$ is a heuristic that quantifies the amount of information lost by the imputation procedure used. $\textsc{Time}(q) \in (0,\infty)$ is an estimate of the runtime of a query q derived from table statistics, selectivity estimation of the query predicates, and the complexity of the impute operation.

3.4.1 Pareto-optimal plans

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

$$Q = \{q \mid \nexists q' \in S.\ q \neq q' \wedge q' \succ q\}.$$

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

This set Q will be the Pareto frontier [19] of S, and it contains the best option for all feasible trade-offs of TIME and PENALTY for the current query. This set of plans is significantly smaller than the set of all plans, which our algorithm prunes away.

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 PENALTY value that the user is willing to tolerate relative to the minimum possible among the frontier plans.

DEFINITION 2. *Let Q be a set of plans. For $q \in Q$, we say q is α-bound if*

$$\textsc{Penalty}(q) - \min_{q' \in Q} \textsc{Penalty}(q') \leq \alpha.$$

We denote this as q^α. Let the set of α-bound plans in Q be Q^α. We say a plan q^α is α-bound optimal if

$$\textsc{Time}(q^\alpha) = \min_{q' \in Q^\alpha} \textsc{Time}(q').$$

Given α and Q, ImputeDB returns an α-bound optimal plan in Q. 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 lowest runtime (at the expense of quality).

3.4.2 Cardinality estimation

The computation of $\textsc{Penalty}(q)$ and $\textsc{Time}(q)$ relies on accurate cardinality estimates for the plan q and its sub-plans. These estimates are impacted not just by filtering or joining, as in the traditional relational algebra, but also by the imputation operators.

To compute cardinality, we maintain histograms for each column in the database. These histograms track the distribution of values in the column as well as the number of missing values. 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 described in Algorithm 1 to account for the new operation in the plan.

Drop operators (Lines 3 to 6) reduce the cardinality of their input relation by removing tuples which contain missing values. For a *Drop* δ_C, for each column $c \in C$ we set its missing value count to zero, because all of these tuples will be dropped. This can result in a set of histograms which have different cardinalities if some columns have more missing values than others. To account for the discrepancy, we rescale all the histograms without changing the shape of their distributions to match the lowest cardinality column.

Impute operators (Lines 7 to 11) maintain the cardinality of their inputs, but they also fill in missing values. For an *Impute* μ_C, we set the missing value count to zero, and rescale each histogram so that its new cardinality, with no missing values, is the same as its old cardinality. We assume that the imputed values will not change the shape of the distribution of values in each column.

Cardinality estimation for standard relational operators (Lines 12 to 16) is the same as in a typical database.

Algorithm 1 An algorithm for in-plan histogram updates

```
Input:
  • $H$: A map from attribute names to histograms.
  • $op$: A relational operator.
 1: function UpdateHistograms($H$, $op$)
 2:   Let $Cols$ be the set of attribute names in $H$.
 3:   if op = $\delta_C$ then
 4:     Set null count to zero for $H[c]$ for $c \in C$.
 5:     $m \leftarrow \min_{d \in Cols} \textsc{Card}(H[d])$
 6:     Scale $H[d]$ s.t. $\textsc{Card}(H[d]) = m$ for $d \in Cols$.
 7:   else if op = $\mu_C$ then
 8:     for $c \in C$ do
 9:       $k \leftarrow \textsc{Card}(H[c])$
10:       Set null count of $H[c]$ to zero.
11:       Scale $H[c]$ s.t. $\textsc{Card}(H[c]) = k$.
12:   else if op = $\sigma_\phi$ then
13:     Scale $H[d]$ by $\textsc{Selectivity}(\phi, H)$ for $d \in Cols$.
14:   else if op = $\bowtie_\phi$ then
15:     Let $k_{join}$ be the estimated cardinality of the join.
16:     Scale $H[d]$ s.t. $\textsc{Card}(H[d]) = k_{join}$ for $d \in Cols$.
17:   return $H$
```

3.4.3 Imputation quality

Each imputation method has an associated cost function P, which is a measure of the quality of its output. ImputeDB estimates the aggregate quality of the imputations in a plan q using the heuristic measure $\textsc{Penalty}_P(q)$. Underlying this is a helper function, $\mathrm{L}_{P(q)}$, which is a recursive accumulator of penalties along imputation nodes instantiated with an impute-specific penalty operator. This accumulator is computed as follows.

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

$$\textsc{Penalty}_P(q) = \frac{\mathrm{L}_P(q)}{\mathrm{L}_{P_\delta}(q)}$$

$\textsc{Penalty}_P(q)$ aggregates penalties from individual imputations to produce a measure of the quality of imputation for the entire plan. For imputation on a subplan q', a penalty $P(q')$ in (0,1] is assigned to $\mu(q')$ and a penalty of 1 is assigned to $\delta(q')$. The function $\mathrm{L}_P(q')$ is then defined recursively to aggregate these penalties for the entire plan. The result is normalized by $\mathrm{L}_{P_\delta}(q')$, which is the total penalty for always dropping tuples with NULL values and is equal to the number of imputation operators in the plan.

Constructing P: The form of P is chosen to reflect the properties of the imputation method being used. A well-chosen P should provide a reasonable estimate of the quality of imputation from the set of plan histograms. From these, P can obtain the number of clean and dirty attributes in its input, their cardinalities, and approximations of distribution statistics such as mean and variance. Intuitively, imputation quality *increases* when more complete attributes and complete tuples are available; correspondingly, $P(q')$ *decreases* as the number of values increases. The penalty should also reflect the intuition that the number of training observations has diminishing returns by decreasing more slowly as the number of values increases. Finally, the penalty may increase as the amount of noise or spread in the data increases, as this makes it more difficult to obtain precise imputations.

Imputation quality $P(q') = 0$ is unattainable, as this suggests the values of missing elements are known exactly. On the other hand, $P(q') = 1$ occurs when all tuples with missing values are dropped, as this represents the worst case in terms of recovering information from these tuples. In Section 3.5, we discuss choices for P for several common imputation methods.

3.4.4 Query runtime

To compute $\text{Time}_T(q)$, 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. We assume that I/O costs dominate CPU costs for relational operators. A database implementation for which this assumption is not true, e.g. an in-memory database, will need a different cost model. 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 never require repeated tuple fetches and have low computational overhead so we assume that they take negligible time.

We extend these heuristics to account for the new operators: *Drop* and *Impute*. *Drop* is a special case of a filter, so its time complexity is correspondingly negligible.

Constructing T: As with Penalty, the time computation for *Impute* depends on the properties of the underlying algorithm via a function T. If we assume that the *Impute* routine is CPU-bound, then we adjust T by some constant factor f, thereby scaling the result to be comparable with I/O-bound routines.

The form of T depends on the specific imputation method, but there are several considerations in choosing T. In particular, T should *increase* as the number of tuples and complete and dirty attributes passed to the imputation function increases, and it should take into account the complexity of the imputation algorithm. Analysis of the time complexity of different imputation algorithms, especially with in-database execution, is beyond the scope of this paper. As with P, it is the ordering between queries imposed by T that is more important than the precision of the estimates.

Note that for every imputation operator inserted into the query plan, a new statistical model is instantiated and trained. Though it is tempting to try to pre-train imputation models, the working data, set of complete attributes, and set of attributes to impute are unknown until runtime, making this strategy infeasible. Thus, if applicable, T should take into account both training and application of the imputation model.

3.5 Imputation Strategies and their API

ImputeDB is designed so that any imputation strategy can be plugged in with minimal effort and without changes to the optimizer, by supplying penalty (P) and time (T) functions (Section 3.4) pertinent to the algorithm. These cost functions are then used by the optimizer during planning. The flexibility of this approach is in the spirit of query optimization, which aims to reconcile performance with a declarative interface to data manipulation. To motivate this API, we describe how a developer might specify these functions for a general-purpose imputation algorithm, as well as for two simpler imputation methods.

3.5.1 Decision trees

As a reference implementation, ImputeDB uses a general-purpose imputation strategy based on chained-equation decision trees. Chained-equation imputation methods [5] (sometimes called *iterative regression* [12]) impute multiple missing attributes 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 effective [2] and are widely used in epidemiological domains [4].

The chained-equation algorithm proceeds by iteratively fitting decision trees for the target attributes. In each iteration, the missing values of a single attribute are replaced with newly imputed values. Attributes that contain missing values but that are not being imputed are ignored by the imputation. Values in a single column can be updated over multiple iterations of the algorithm. With more iterations, the quality of the imputation improves as values progressively reflect more accurate relationships between attributes. The algorithm terminates after a fixed number of iterations, or earlier if convergence is achieved, indicated by unchanged values in a column in subsequent iterations. For our experiments, we use Weka's `REPTree` decision tree implementation [25], a variant of the canonical `C4.5` algorithm [22]. Roughly, the time complexity of building one decision tree is $\mathcal{O}(nm\,log(n))$, where n is the number of tuples and m is the number of dependent attributes [25].

Recall that C is the set of dirty attributes to be imputed, where $C \subseteq \text{Dirty}(q)$. With k cycles of the algorithm, where in each cycle there are $|C|$ iterations of fitting models for each dirty attribute, we have

$$\begin{aligned} T(q) = k \times |C| \times \text{Card}(q) \times \\ (|\text{Clean}(q)| + |C| - 1) \times \log \text{Card}(q). \end{aligned}$$

Separately, we specify a heuristic penalty that takes into account the improvement in imputation quality from more tuples and attributes, as well as the diminishing returns from more data.

$$P(q) = \frac{|C|}{\sqrt{|\text{Attr}(q)| \times \text{Card}(q)}}$$

3.5.2 Non-Blocking Approximate Mean

At the other end of the complexity spectrum, mean value imputation is a common choice for exploratory data analysis. Mean value imputation replaces missing values in a column with the mean of the available values in that column.

We can take advantage of the histograms maintained by ImputeDB to estimate the mean and construct a non-blocking impute operator. Applying the heuristics from Section 3.4.3 and Section 3.4.4, we have

$$P(q) = \sum_{c \in C} \frac{1 + \widehat{Var}(c)}{\text{Card}(q)}$$
$$T(q) \approx 0.$$

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

$$\begin{aligned} \llbracket Q[T] \lhd p \rrbracket &= Q \cup \{(T, P)\} \text{ where} \\ P &= \{q \in S \mid \textsc{Dirty}(q) \neq \textsc{Dirty}(p) \vee \nexists q' \in S\colon q' \succ q\} \\ S &= \llbracket Q[T] \rrbracket \cup \{p\} \end{aligned}$$

Figure 4: Semantics of the plan cache access and insertion syntax. We use $\llbracket\ \rrbracket$ to mean evaluation. Q is a plan cache, T is a set of tables, and p is a query plan. The cache guarantees that plans for the same set of tables either have distinct dirty sets or are Pareto optimal.

Note that P increases with the variance of the columns and decreases with their cardinality, reflecting the intuition that the quality of the mean is lower with high variance and higher with more data.

3.5.3 *Hot deck*

Random hot deck imputation is a non-parametric method in which missing values in a column are replaced with randomly selected complete values from the same column, preserving the distribution of the complete values. We use the same P as for mean value imputation, but we modify T to account for the fact that hot-deck is blocking in our implementation.

$$P(q) = \sum_{c \in C} \frac{1 + \widehat{Var}(c)}{\textsc{Card}(q)}$$
$$T(q) = \textsc{Card}(q)$$

3.6 Query Planning

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

3.6.1 *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 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. The plan-cache semantics are shown in Figure 4.

The plan cache uses the partial order of plans defined by $\langle \textsc{Dirty}, \textsc{Penalty}, \textsc{Time} \rangle$ to collect sound and complete Pareto frontiers. Plans with different $\textsc{Dirty}$ sets cannot be compared. The final Pareto frontier produced by the optimizer is the set of plans with the best imputation quality and runtime trade-off.

3.6.2 *Planning algorithm*

The planner (Algorithm 2) 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 7–Line 8). This set will be used to determine which attributes are imputed. Next, it constructs a plan cache. For each table used in the query, any selections are pushed to the leaves, and a search over imputation operations produces various possible plans, which are added to the cache (Line 12). If no selections are available, a simple scan is added to the plan cache (Line 13). Join order is jointly optimized with imputation placements and a set of plans encompassing all necessary tables is produced (Line 14). This set is then extended for any necessary grouping and aggregation operations (Line 22) and for projections (Line 24). 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 [3], 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.

Algorithm 2 A query planner with imputations.

```
Input:
  • $q$: A query plan.
  • $C_l$: A set of attributes that must be imputed in the output of this query plan.
  • $C_g$: The set of attributes which are used in the final plan.
 1: function $\textsc{Impute}(q, C_l, C_g)$
 2:     $D_{must} \leftarrow \textsc{Dirty}(q) \cap C_l$
 3:     $D_{may} \leftarrow D_{must} \cup (\textsc{Dirty}(q) \cap C_g)$
 4:     $Q \leftarrow (\{q\}$ if $D_{must} = \varnothing$ else $\{\mu_{D_{must}}(q), \delta_{D_{must}}(q)\})$
 5:     return $(Q$ if $D_{may} = \varnothing$ else $Q \cup \{\mu_{D_{may}}(q), \delta_{D_{may}}(q)\})$

Input:
  • $T$: A set of tables.
  • $F$: A $T \times \Phi$ relation between tables and filter predicates.
  • $J$: A $T \times \Psi \times T$ relation between tables and join predicates.
  • $P$: A set of projection attributes.
  • $G$: A set of grouping attributes.
  • $A$: An aggregation function.
  • $\alpha$: A parameter in $[0,1]$ that expresses the trade-off between performance and imputation quality.
 6: function $\textsc{Plan}(T, F, J, P, G, A, \alpha)$
 7:     $C_g \leftarrow \bigcup_{\psi \in J} \textsc{Attr}(\psi)$          ▶ Collect relevant attributes.
 8:     $C_g \leftarrow C_g \cup \bigcup_{\phi \in F} \textsc{Attr}(\phi) \cup P \cup G \cup \textsc{Attr}(A)$

 9:     Let $Q$ be an empty plan cache.
10:     for $t \in T$ do          ▶ Add selections to the plan cache.
11:         if $\exists \phi : (t, \phi) \in F$ then
12:             $Q[\{t\}] \lhd \{\sigma_\phi(q) \mid q \in \textsc{Impute}(t, \textsc{Attr}(\phi), C_g)\}$
13:         else $Q[\{t\}] \lhd \{t\}$

14:     for $size \in 2...|T|$ do          ▶ Optimize joins.
15:         for $S \in \{$all length $size$ subsets of $T\}$ do
16:             for $t \in S$ do
17:                 for $(t, \psi, t') \in J$ where $t' \in S \setminus t$ do
18:                     $L \leftarrow \{\textsc{Impute}(q, \textsc{Attr}(\psi), C_g) \mid q \in Q[S \setminus t]\}$
19:                     $R \leftarrow \{\textsc{Impute}(q, \textsc{Attr}(\psi), C_g) \mid q \in Q[\{t\}]\}$
20:                     $Q[S] \lhd \{l \bowtie_\psi r \mid l \in L, r \in R\}$

21:     $B \leftarrow Q[T]$          ▶ Get the best plans for all tables.
22:     if $G \neq \varnothing$ then          ▶ Add optional group & aggregate.
23:         $C_l \leftarrow G \cup \textsc{Attr}(A)$
24:         $B \leftarrow \bigcup_{q \in B} \{g(q', G, A) \mid q' \in \textsc{Impute}(q, C_l, C_g)\}$
25:     $B \leftarrow \bigcup_{q \in B} \{\pi_P(q') \mid q' \in \textsc{Impute}(q, P, P)\}$
26:     return $p \in B$ s.t. $p$ is $\alpha$-bound optimal.
```

3.7 Complexity

Our optimization algorithm builds off the approach taken by a canonical cost-based optimizer [3]. 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 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 dramatically.

Finally, we note that for the queries we have examined, the 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.

In order to support a large number of joins, with lower planning times, ImputeDB can maintain approximations of the Pareto sets. A non-dominated plan can be approximated by an existing plan in the set, if the distance between them is smaller than some predefined bound. Approximated plans are not added to the frontier, pruning the search space further. Planning times across both exact and approximate Pareto frontiers are discussed further in Section 5.3.

4. IMPLEMENTATION

Adding imputation and our optimization algorithm to a standard SQL database requires modifications to several key database components. In this section we discuss the changes required to implement dynamic imputation. We based this directly on our experience implementing ImputeDB on top of SimpleDB [1], a teaching database used at MIT, University of Washington, and Northwestern University, among others.

- **Extended histograms:** Ranking query plans that include imputation requires estimates of the distribution of missing values in the base tables and in the outputs of sub-plans. We extend standard histograms to include the count of missing values for each attribute. On the base tables, this provides an exact count of the missing values. In sub-plans we use simple heuristics—as discussed in Section 3.4.2—to estimate the number of missing values after applying both standard and imputation operators. These cardinality estimates are critical to the optimizer's ability to compare the performance and imputation quality of different plans.
- **Dirty sets:** The planner needs to know which attributes in the output of a plan might contain missing values so it can insert the correct imputation operators. To provide the planner with this information, we over-approximate the set of attributes in a plan that may have missing values. Each plan has an associated *dirty set*, described in Section 3.3, which tracks these attributes.
- **Imputation operators:** We extend the set of logical operations available to the planner with the *Impute* and *Drop* operators. The database must have implementations for both operators and be able to correctly place them while planning. In addition to the normal heuristics for estimating query runtime, these operators have a cost function PENALTY (Section 3.4.3) which estimates the quality of their output. The planner must be able to optimize both cost functions and select an appropriate plan from the set of Pareto-optimal plans.
- **Integrated optimizer:** We believe that it would be relatively simple to extend an existing query optimizer to insert imputations immediately after scanning the base tables. However, integrating the imputation placement rules into the optimizer allows us to explore a large space of query plans which contain imputations. In particular, this tight integration allows us to jointly choose the most effective join order and imputation placement.

5. EXPERIMENTS

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

5.1 Data Sets

We collect three data sets for our experiments. For all data sets, we select 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.

5.1.1 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 [6]. The data consists of survey responses, physical examinations, and laboratory results, among others.

There are 6 tables in the NHANES data set. We use three tables for our experiments: demographic information of subjects (`demo`), physical exam results (`exams`), and laboratory exam results (`labs`).

The original tables have a large number of attributes, in some cases providing more granular test results or alternative metrics. We focus on a subset of the attributes for each table to simplify the presentation and exploration of queries. Table 2 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.

5.1.2 freeCodeCamp 2016 New Coder Survey

For our second set of experiments, we use data collected by freeCodeCamp, an open-source community for learning

Table 2: Percentage of values missing in the CDC NHANES 2013–2014 data.

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

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 %

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

(c) Physical Results (`exams`). 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 %

Table 3: Percentage of values missing in the freeCodeCamp Survey data (`fcc`).

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 %

to code, as part of a survey of new software developers [11]. 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, *bootcamploanyesno*, 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 3 along with the percentage of missing values.

5.1.3 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, which we then artificially dirty by replacing 40% of the values uniformly at random with NULL values. The final dataset consists of 671,153 rows and 37 integer columns.

5.2 Queries

We collect a set of queries (Table 4) 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 benchmark queries performed well for both cases of `AVG` and `COUNT` aggregates, so we expect ImputeDB to perform similarly for `SUM`, with errors proportional to the fraction of relevant tuples retrieved and the `AVG` value estimated. Aggregates such as `MAX` and `MIN` are unlikely to perform well with dynamic imputation, a difficulty carried over from traditional imputation and compounded by the dynamic nature of our approach.

5.3 Results

We evaluate ImputeDB optimizing for quality ($\alpha = 0$), optimizing for performance ($\alpha = 1$), and targeting a balance of each ($\alpha = 0.5$). As a baseline, we fully impute the tables used by each query and then run the query on the imputed tables, simulating the process that an analyst would follow to apply existing imputation techniques.

Runtime vs. Base-table Imputation Figure 5 summarizes the performance results. The quality-optimized queries, with runtimes ranging from 75 ms to 1 second, are an order-of-magnitude faster than the baseline, with runtimes ranging from 6.5 seconds to 27 seconds. We get another order-of-magnitude speedup when optimizing for performance, achieving runtimes as low as 12 ms to 19 ms. These speedups, ranging from 10x to 1400x depending on α and query, constitute significant workflow improvements for the analyst.

Accuracy vs. Base-table Imputation (AVG) Table 5 shows the Symmetric-Mean-Absolute-Percentage-Error (SMAPE) [17] for ImputeDB's query results compared to the baseline, using the chained-equation decision trees model. This measures the error introduced by on-the-fly imputation as compared to full imputation on the base tables. This comparison is the relevant one in practice — an analyst would only be considering the trade-offs in imputation quality and time for dirty input data when the ground truth is unavailable.

To calculate SMAPE for each α, we compute tuple-wise absolute percentage deviations within each iteration of each query, and average this value over all iterations. Relative errors

Table 4: Queries used in our experiments.

#	Queries on CDC data
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 >= 13AND 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;`
#	Queries on freeCodeCamp data
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;`

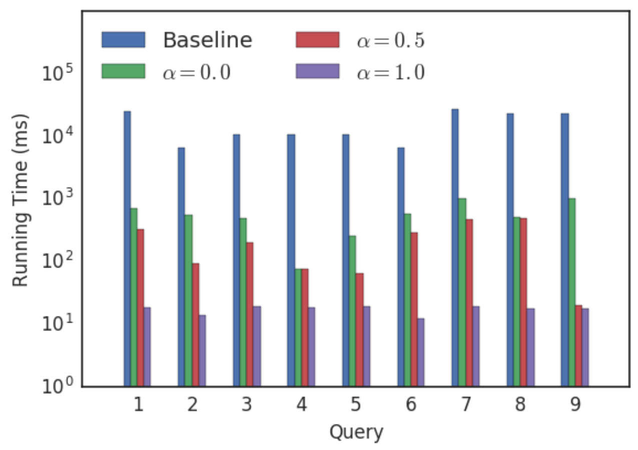


Figure 5: Runtimes for the queries in Table 4 using reference chained-equation decision trees algorithm, for various settings of α. Each query and each value of α was run 200 times.

of query results, when optimizing for quality ($\alpha = 0$), are low — between 0 and 8% — indicating that on-the-fly imputation produces results similar to the baseline. When optimizing for performance ($\alpha = 1$), relative errors can be modestly higher — up to 20% in the worst case, as this approach most closely corresponds to dropping all NULLs. Thus, it is important to recognize the trade-offs between quality and performance for a specific use case.

Accuracy vs. Base-table Imputation (COUNT) We calculate the number of tuples used to produce each aggregate output. The count fraction columns in Table 5 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 due to insertion of δ operators. Even in cases where the SMAPE reduction from $\alpha = 1$ and $\alpha = 0$ is small (Query 2 and 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 completely 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.

We can also trivially extend ImputeDB to warn users when the query chosen for execution has a high Penalty estimate, along with the number of tuples that have been dropped (after execution), so that situations with high potential for skewed results can be identified by the user.

Alternate Imputation Methods We experiment with mean-value imputation and hot deck imputation (Section 3.5). In our implementation of mean-value imputation, to facilitate non-blocking operation within the iterator model, we estimate the column mean from base table histograms. Therefore, imputed values are identical no matter where the operator is placed in the query plan (relative errors are 0). Runtimes are in the range of 12 ms to 25 ms for queries using ImputeDB, as compared to 24 ms to 44 ms when imputing on the base table. Here, the time cost of the imputation is low no matter how many tuples need to be imputed.

In hot deck imputation, runtimes are also similar across α, ranging from 12 ms to 29 ms using dynamic imputation, as compared to 21 ms to 92 ms when imputing on the base table. In this case, the cost of buffering tuples and sampling at random is negligible compared to the rest of query execution. Relative errors using hot deck are close to zero, ranging from 0–3%. (The exception is Query 8 with $\alpha = 1$, which has error of 24% and exhibits significant skew.) Since the baseline, in these cases, is hot deck imputation on the base table, it is skew in the distributions arising from filters and joins that often leads to larger errors. Indeed, ImputeDB yields larger benefits when using relatively sophisticated, higher quality

Table 5: Symmetric-Mean-Absolute-Percentage-Error for queries run under different α parameterizations, as compared to the baseline. Queries optimized for quality ($\alpha = 0$) generally achieve lower error than queries optimized for efficiency ($\alpha = 1$). With ($\alpha = 0.5$), an intermediate plan may be chosen if available. 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 fraction reflects more potential for errors.

	SMAPE			Count Fraction		
Query	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$	$\alpha = 0$	$\alpha = 0.5$	$\alpha = 1$
1	0.39	1.19	1.24	1.00	0.89	0.88
2	1.49	1.65	2.41	1.00	0.99	0.33
3	1.05	1.05	2.14	1.00	1.00	0.57
4	0.22	0.30	0.30	0.96	0.89	0.80
5	0.02	0.19	0.20	1.01	0.94	0.94
6	1.04	0.88	4.43	0.79	0.80	0.64
7	0.03	0.43	0.40	1.00	0.66	0.65
8	7.97	7.97	19.99	1.00	1.00	0.22
9	2.03	2.17	2.49	1.00	0.85	0.84

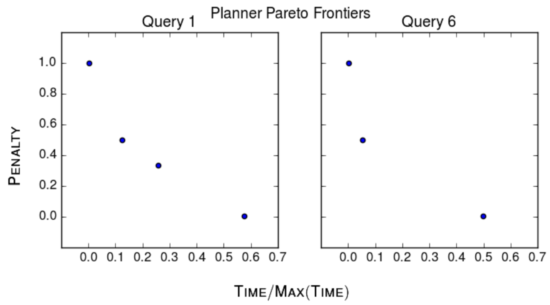


Figure 6: ImputeDB's final planner Pareto frontier for Query 1 and Query 6. For clarity, we have not shown plans that provide nearly indistinguishable trade-offs in either dimension.

imputation algorithms as opposed to simpler strategies that would be quick to implement even on the base tables.

Pareto Frontiers Figure 6 shows the final Pareto frontiers produced by the planner for Query 1 and Query 6. These are the resulting set of plans after pruning dominated plans throughout our algorithm and encapsulate the possible trade-offs that the user can make in choosing a final plan for execution. The frontier is available at the end of the planning stage and can be exposed to the user to guide their workflow.

Planning Times Planning times for our benchmark queries are reasonable, with mean planning times across the benchmark queries ranging from 0.7 ms to 2.8 ms, and ranging from 0 to 14% of total runtime. In 99% of individual cases, the optimizer returned a query plan within 4 ms. The choice of α has no impact on planning times, because query selection is performed after collecting the Pareto frontier.

Approximate Pareto Sets However, ImputeDB's planning algorithm is still exponential in the number of joins in the plan, and the use of Pareto sets exacerbates this exponential growth. We explored the limits of our planning algorithm by constructing a series of queries with increasing number of joins over the CDC tables. Planning times for queries involving 1–5 joins—a practical value for real world exploratory queries—are less than 1 second. We extended ImputeDB to support an approximation of Pareto sets to improve planning times for queries with 6 or more joins. Approximate Pareto sets displayed a reduction in planning time linear with respect to the reduction in size of the final frontier. For queries with 6–8 joins, approximate sets achieved a reduction in planning time between 14% and 23%, on average. For 6 joins, this is a decrease from a mean planning time of 3.3 seconds to 2.5 seconds. For 7 joins, the decrease was from 14.2 to 12.2 seconds, while for 8 joins, it was from 60 to 49 seconds. Users can increase the level of approximation to reduce planning times further.

Imputation on Large Data In many real-world cases, applying imputation to an entire dataset is prohibitively expensive. For example, if we run chained-equation decision trees imputation on the full ACS dataset, it completes in 355 minutes.

In contrast, ImputeDB executes the query `SELECT AVG(c0) FROM acs` over the ACS dataset (also using chained-equation decision trees imputation) in 4 seconds when optimizing for quality and 1 second when optimizing for runtime. Given the runtime difference, we can run approximately 5,000 queries before taking more time than full imputation. The performance difference comes from the fact that ImputeDB only needs to perform imputation on the single column required by the query. Adding selection predicates would further reduce the query runtime by reducing the amount of required imputation. An analyst could do the same, but tracking the required imputations would get increasingly complicated as the queries became more complex. This highlights the benefit of using our system for early data exploration.

6. RELATED WORK

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

6.1 Missing Values and Statistics

Imputation of missing values is widely studied in the statistics and machine learning communities. Missing data can appear for a variety of reasons [12]. It can be missing completely 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, Burgette and Reiter [4] use iterative decision trees for imputing missing data.

The computational difficulties of imputing on large base tables are well-known and can limit approaches. For example, Akande, Li, et al. [2] find that one approach (MI-GLM) is prohibitively expensive when attempting to impute on the American Community Survey dataset. In this case, the complexity comes from the large domains of the variables in the ACS data (up to 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 datasets which are too large to fully impute quickly. 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.

6.2 Missing Values and Databases

There is a long history in the database community surrounding the treatment of NULL. Multiple papers have described various (and at times conflicting) treatments of NULLs [7, 13]. ImputeDB's design invariants eliminate 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. Hellerstein [14] surveys the methods and systems related to missing value detection and correction.

Wolf, Khatri, et al. [26] directly treat queries over databases with missing values 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 a physical process like sensor error is the cause of missing values, values can be imputed with high confidence. Fernández-Moctezuma, Tufte, et al. [9] use feedback punctuation 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 [18] 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, among other broader functionality. Experts can further customize strategies and incorporate 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.

Yang, Meneghetti, et al. [27] develop an incremental approach to ETL data cleaning, through the use of probabilistic compositional repair operators, called *Lenses*, which balance quality of cleaning and cost of those operations. These operators can be used to repair missing values by imposing data constraints such as `NOT NULL`. However, the model used to impute missing values must be pre-trained and all repairs are done on a per-tuple basis rather than at the plan-level.

ImputeDB's cost-based query planner is partially based on the seminal work developed for System R's query planning [3]. 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.

The planning algorithm that we present has similarities to work on multi-objective query optimization [24]. Both algorithms handle plans which have multiple cost functions by maintaining sets of Pareto optimal plans and pruning plans which become dominated. In ImputeDB, costs are expected to grow monotonically as the size of the plan increases, which simplifies the optimization problem significantly. Trummer and Koch [24] handle the more complex case of cost functions which are piecewise linear, but not necessarily monotonic.

6.3 Forecasting and Databases

Parisi, Sliva, et al. [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 subsequent 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.

Fischer, Dannecker, et al. [10] 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, and is not 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.

Duan and Babu [8] describe 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. As with 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, and considers the trade-offs between accuracy and computation time.

7. CONCLUSION

Our implementation of ImputeDB and experiments show that imputation of missing values can be successfully integrated into the relational algebra 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, running standard SQL queries over tuples 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 (0-20% error, Section 5.3). 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 statistics and database communities. In contrast to existing work in statistics, 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, operations have not been integrated into the planner. The novelty of our contribution lies in the formalization of missing value imputation for query planning, resulting 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.

7.1 Future Work

ImputeDB opens up multiple avenues for further work.

- **Imputation confidence:** Obtaining confidence measures for query results produced when missing values are imputed is a possible improvement to the system. These measures could be obtained in general using resampling methods like cross-validation and bootstrapping [16]. In contrast to standard confidence calculations, 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. This goes beyond simply caching imputations, 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.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge Akande, Li, and Reiter for giving us a cleaned copy of the 2012 ACS PUMS. This work was supported in part by the Qatar Research Computing Institute.

References

[1] *6.830 Lab 1: SimpleDB.* URL: `http://db.csail.mit.edu/6.830/assignments/lab1.html`.

[2] O. Akande, F. Li, et al. "An Empirical Comparison of Multiple Imputation Methods for Categorical Data". In: *arXiv:1508.05918* (2015).

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

[4] 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.

[5] S. van Buuren and K. Groothuis-Oudshoorn. "mice: Multivariate Imputation by Chained Equations in R". In: *Journal of Statistical Software* 45.3 (2011), pp. 1–67.

[6] 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.

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

[8] S. Duan and S. Babu. "Processing Forecasting Queries". In: *VLDB '07*. VLDB Endowment, 2007, pp. 711–722.

[9] R. Fernández-Moctezuma, K. Tufte, et al. "Inter-Operator Feedback in Data Stream Management Systems via Punctuation". In: *CIDR*. 2009.

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

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

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

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

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

[15] W. Kim, B.-J. Choi, et al. "A Taxonomy of Dirty Data". In: *Data Mining and Knowledge Discovery* 7.1 (Jan. 1, 2003), pp. 81–99.

[16] 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.

[17] 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.

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

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

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

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

[22] J. R. Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993.

[23] D. B. Rubin. "Inference and Missing Data". In: *Biometrika* 63.3 (1976), pp. 581–592.

[24] I. Trummer and C. Koch. "Multi-Objective Parametric Query Optimization". In: *VLDB* 8.3 (2014), pp. 221–232.

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

[26] G. Wolf, H. Khatri, et al. "Query processing over incomplete autonomous databases". In: *VLDB '07*. VLDB Endowment. 2007, pp. 651–662.

[27] Y. Yang, N. Meneghetti, et al. "Lenses: An on-demand approach to etl". In: *VLDB* 8.12 (2015), pp. 1578–1589.