ImputeDB:
Data Imputation as a Query Optimization

Jose Cambronero John K. Feser Micah Smith

September 21, 2017

Abstract

In order to study the placement of an imputation step, we create a logical imputation operator (along with respective physical instances) and incorporate its placement as part of the query plan optimization process in SimpleDB. We introduce measures for information loss and runtime for imputation operations, which outline the main trade-offs in the imputation placement. We add these measures into our cost estimation, allowing us to intelligently place the data imputation step during query planning. We show the trade-offs between efficiency and accuracy for simple data imputation models.

1 Introduction

Handling incorrect or dirty data is a complex and challenging problem for data scientists. One way in which a dataset can be dirty is for parts of it to be missing altogether. Missing data is one of the simpler variants of dirty data, but if handled naively, can still cause analyses to be incorrect. To handle this problem, users may manually clean their dataset by performing some statistical analysis to replace missing data elements with likely values. This process is called *imputation*. Traditional imputation methods replace all missing values in a dataset to create a new clean dataset. Although manual imputation solves the problem of missing data, in the age of big data it may be very expensive to run an imputation algorithm on an entire dataset. Additionally, it may not even be necessary to completely clean the data to make it usable. Some users may be willing to run queries on dirty data, simply ignoring the missing values, as long as they do not have to pay the cost of imputation. Others may want to run queries on a subset of the data, and so do not need to impute every field in every record. Yet others may want to customize the imputation algorithm for the tradeoffs and demands of a particular domain.

In this paper, we present ImputeDB, a database system which is designed to interact with a dirty dataset as though it were clean. The guiding design principle behind ImputeDB is that the user should never see missing data or have to modify their queries to account for it. To achieve this goal, we perform imputation on the fly, during query execution. Performing imputation at query time allows our system to impute only the data necessary to run the query, and it allows users to flexibly trade accuracy for computation time.

2 Related Work

2.1 Missing Values and Statistics

Imputation of missing values is a widely studied field within the statistics and machine learning communities. As highlighted in [1], missing data can appear for a variety of reasons, including both random and conditioned on existing values (observed and missing). Methods in the statistical community focus on correctly modeling relationships between the attributes to factor in varied forms of missingness. For example, Burgette and Reiter ([2]) discuss the usage of sequential regression trees for imputing missing data.

In [3], Akande et al analyze the performance of various multiple imputation techniques on the American Community Survey dataset (Sec 5.1). The computational difficulties of imputing on large base tables are well known and can limit approaches. For example, Akande 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 tradeoff between information loss and computational complexity (in terms of time). 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.

2.2 Missing Values and Databases

There is a long history in the database community surrounding the treatment of nulls. As early as 1973, [4] provides a treatment of the semantics of null. Multiple papers have described various (at times conflicting) treatments of nulls [5]. ImputeDB's main design invariant - that no relational operator see attributes with missing values for attributes it must operate on and users should never see missing data - eliminates the need to handle null value semantics, while guaranteeing soundness (modulo imputation strategy) of the query evaluation.

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 [6].

BayesDB [7] provides users with a simple interface to leverage statistical inference techniques in a database. Non-experts can use a simple declarative language, akin to 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 perform 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[8]. However, in contrast to System R, ImputeDB performs additional histogram transformations to account for the changing nature of missing values.

3 Examples

Consider the case of an analyst looking to explore the sources and causes of polling error in a presidential election. One hypothesis [9] is that the rapid switch from landlines to cellphones increases the difficulty of contacting potential poll respondents. The analyst may begin her analysis by comparing the polling error in a state with the distribution of households with landlines, for the set of states for which polling error was most pronounced. In order to probe deeply, the analyst accesses a household-level survey that includes data on whether the household has a landline. One such survey — which will be discussed further (Sec 5.1) — is the American Community Survey (ACS), conducted by the US Census Bureau.

Surveys are rife with missing values due to non-response and other issues, and the ACS is no exception. The Census Bureau and other similar organization attempt to present data in as unadulterated a fashion as possible. The ACS dataset is not amenable to one-time imputation by either the Census Bureau, for this reason, or the analyst's organization, which likely uses the same database for a variety of unrelated uses (asking for different imputation strategies). Thus, the analyst must deal with missing values in one way or another.

In this case, a non-negligible fraction of respondents omitted information on whether they have a landline. The analyst hypothesizes that older houses are more likely to have landlines and that households that are relatively new are less likely to have landlines. The analyst could take advantage of these presumed correlations to impute the missing values on the base table in one expensive operation. On the other hand, the query could be run directly and rows with missing values may be dropped entirely. With ImputeDB, the analyst could impute the relevant subset of data on-the-fly, achieving better performance without losing the information in the dropped examples.

The analyst submits a query (Figure 1) to ImputeDB which finds an optimized query plan[1]. The resulting query plan (Figure 2) imputes missing values for `acs.TEL` after states with high polling error have been selected. This decision reduces the set of tuples that must be input into the imputation algorithm to just those that are relevant to the query. Later, we show the algorithms that the ImputeDB query optimizer uses in order to detect these opportunities and control the tradeoffs between efficiency and imputation quality. Finally, note that the

[1]The analyst specifies a relatively high value of α, the quality emphasis parameter, in order to receive a plan that does an expensive imputation step. See Section 4.3 for more details.

```
SELECT polling.ST, AVG(acs.TEL)
FROM polling, acs
WHERE polling.ST = acs.ST
  AND polling.ERROR > 50          -- 5 percentage points
GROUP BY polling.ST;
```

Figure 1: A typical analyst query on ACS data

optimizer doesn't place any imputation operator for the `acs.ST` column, as it uses a histogram to detect that there are no missing values there (as might be expected for a survey question that isn't impacted by non-responses).

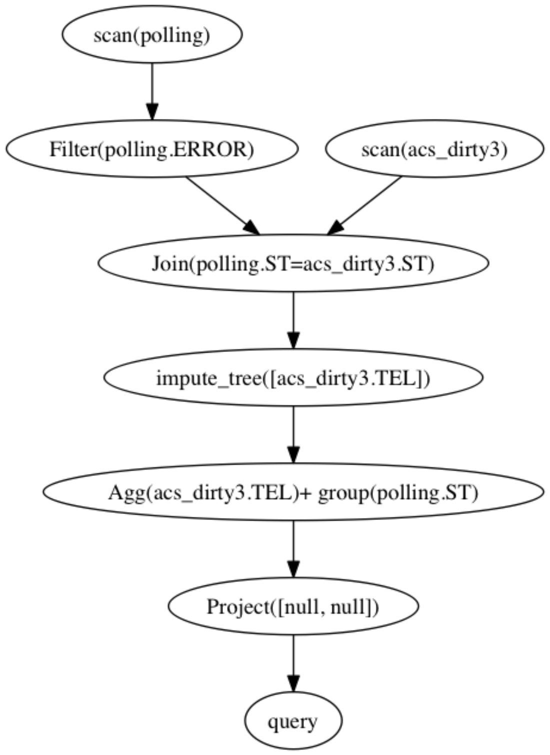
scan(polling)
Filter(polling.ERROR)
scan(acs_dirty3)
Join(polling.ST=acs_dirty3.ST)
impute_tree([acs_dirty3.TEL])
Agg(acs_dirty3.TEL)+ group(polling.ST)
Project([null, null])
query

Figure 2: Query plan generated by ImputeDB prioritizing imputation quality

4 Algorithm

Our optimizer searches a restricted space (Sec. 4.2) of query plans for a plan that minimizes a metric (Sec. 4.3) which combines the runtime cost of the query and the quality of the results. This plan must not emit any tuples which contain null values, regardless of the state of the base tables. Additionally, the traditional relational operators (selection σ, projection π, join $\bowtie$, and group-by/aggregate) must never observe a null value in any attribute that they directly operate on.

4.1 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 in C in the relation R. *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.

4.2 Search space

To reduce the size of the query plan search space, only plans that fit the following template are considered. First, all filters are pushed to the leaves of the query tree, immediately after the scans. Joins are performed after filtering, and only left-deep plans are considered. Any group-by/aggregate will be performed after the final join. Finally, projections are placed at the root of the query tree. The space of query plans is similar to that considered by System R [8], with the addition of imputation operators appearing before/after traditional operators.

4.3 Cost model

We rank query plans Q using a cost model $\text{Cost}(Q) = (1-\alpha) \times \text{Time}(Q) + \alpha \times \text{Loss}(Q)$. $\text{Time}(Q)$ is an estimate of the runtime of the query which is derived from table statistics and selectivity estimation of the query predicates. $\text{Loss}(Q)$ is an estimate of the amount of error introduced by the imputation procedure. α is a parameter to the query optimizer that controls the emphasis on quality over performance. $\alpha = 1.0$ means that the query should be as accurate as possible (at the expense of performance), $\alpha = 0.0$ means that the query should be as fast as possible (at the expense of accuracy).

In order to correctly estimate $\text{Time}(Q)$ and $\text{Loss}(Q)$, the system must have cardinality estimates for each sub-query in each query plan. 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. For simplicity, each of the logical nodes in a query plan points to a set of histograms. When the optimizer creates a new query plan, it copies the histograms of the sub-plans and modifies them as necessary to account for new operation in the plan. Algorithm 3 describes the process of generating new histograms from sub-plans.

Our runtime cost model is the cost model used in SimpleDB, though modern systems have more sophisticated approaches.

For each sub-query, we keep track of the estimated number of null values in a column (MissingCount(c,Q)) by using the histograms associated with the sub-query. We use this estimate to compute Loss(Q) as follows.

$$\text{Loss}(Q)=\begin{cases}\sum_{c\in C}\text{MissingCount}(c,Q') & Q=\delta_C(Q')\\ \frac{1}{\sqrt{|Q'|}}\sum_{c\in C}\text{MissingCount}(c,Q') & Q=\mu_C(Q')\\ \text{Loss}(Q_1')+\text{Loss}(Q_2') & Q=Q_1'\bowtie_\psi Q_2'\\ \text{Loss}(Q') & Q=\sigma_\phi(Q'),Q=\pi_C(Q')\end{cases}$$

Dropping tuples which contain a null value incurs a loss of 1 for each null field dropped. Imputing null fields incurs a loss penalty $p \in (0,1]$ which decreases as the number of tuples available to train the imputation algorithm increases. Intuitively, imputation accuracy *increases* when more complete attributes and complete tuples are available; correspondingly, the information loss *decreases* with more complete values to work with. The inverse square loss can be seen as a heuristic first approximation to the learning guarantees of regression trees. Since we conceive that alternate imputation strategies could be used, the analyst may provide a corresponding loss with a reasonable parameterization.

To compute Time(Q), we retain the heuristics used by SimpleDB for the time complexity of the typical relational operators, and add new formulations for *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 IO cost per page.

Evaluation of the time complexity for *Impute* is thornier, as the properties of different decision tree building algorithms can vary significantly, the underlying IO cost is not easily extracted (especially in the case that the entire stream of tuples does not fit in memory), and the CPU cost dominates (in contrast to the other operators, which may not even consider CPU cost explicitly). For example, [10] find 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). Indeed, 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. In practice, we find that a simple parameterization can be acceptable (i.e. yield intelligent query plans), though the range of α that does promote tradeoffs is condensed.

4.4 Imputation placement

Imputation operators must be placed so that no relational operator receives 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 in the query plan, but to meet the guarantee that no non-imputation operator sees a null value, there are cases where an imputation operator is required. To track these cases, each query plan is associated with a set of dirty attributes D. An attribute c is *dirty* in some relation if the values for c may contain null. We compute a dirty set for each base table using the table statistics, which track the number of null values in each column. If we apply an imputation operator to a relation R with a dirty set D, $\mu_C(R)$ or $\delta_C(R)$, the resulting query has a dirty set $D' = D \setminus C$. Applying a projection $\pi_C(R)$ produces a dirty set $D' = D \cap C$. A join $R_1 \bowtie_\psi R_2$ with dirty sets D_1 and D_2 produces a dirty set $D' = D_1 \cup D_2$. Filters do not change the dirty set.

That is, the dirty set over-approximates the set of attributes that contain null. For example, a filter might remove all tuples which contain null without changing the dirty set, forcing an unnecessary imputation. We choose to over-approximate the dirty set to avoid the possibility of dropping a tuple that contains a null value without explicitly imputing the value or applying a drop operator.

4.5 Query planning

The input to our query planner is a tuple (T, Φ, Ψ, P, G, A): a set of tables T, a set of filter predicates $\phi_t, t \in T$, a set of join predicates $\psi_{(t_1,t_2)}, t_1, t_2 \in T$, a set of attributes P, and an optional set of attributes G and aggregation function $A \in \{\text{Max,Min,Sum,Avg,Count}\}$.

The query planner must select a join ordering in addition to placing imputation operators as described in Section 4.4.

To reduce the search space, we only consider the minimal imputation, the maximal imputation, and the minimal drop. The minimal imputation (resp. drop) only imputes (resp. drops) the columns required by the relational operator immediately following the imputation. The maximal imputation imputes all columns in the relation, regardless of which are required. Algorithm 1 presents a series of helper functions used by the top level planner, shown in Algorithm 2.

4.6 Imputation Strategies

We design our system such that any imputation strategy can be plugged in with minimal effort and without changes to the optimizer, so the strategy can in principle be targeted to a specific domain. The current version of the system uses a general-purpose imputation strategy based on chained-equation classification and regression trees (CE-CART). Chained equation imputation methods [11] (sometimes called *iterative regression* [1]) impute multiple missing attributes by iteratively estimating predictive models of one missing attribute conditional on all complete attributes and the other missing attributes. The individual predictive models can be customized by the analyst to the problem at hand. Decision tree algorithms, like CART, are found to be effective [3] in empirical studies in general-purpose routines and are widely used in epidemiological domains [2]. Though we provide quantitative results for the CE-CART implementation, ImputeDB is designed to be agnostic to the choice of imputation algorithm, and additional algorithms — with associated time and loss functions — could be inserted without any change in design.

These imputation algorithms are rarely used in isolation — rather, the ultimate goal is to perform statistical analysis of the complete data. To this end, the

Algorithm 1 Base algorithms for query planning with imputations.

```
Require: $q$ is a query plan, $C_{req}$ is a set of attributes that must be imputed.
Ensure: Returns a set of query plans such that $\textsc{Dirty}(q') \cap C_{req} = \varnothing$.
  function $\textsc{AddImpute}(q, C_{req})$
    $C_{min} \leftarrow \textsc{Dirty}(q) \cap C_{req}$
    if $\textsc{Dirty}(q) = \varnothing$ then
      return $\{q\}$
    else if $C_{min} = \varnothing$ then
      return $\{\mu_{\textsc{Dirty}(q)}(q), q\}$
    else
      return $\{\mu_{\textsc{Dirty}(q)}(q), \mu_{C_{min}}(q), \delta_{C_{min}}(q)\}$

Require: $Q$ is a set of query plans.
Ensure: Returns an optimal query plan for each distinct dirty set in $Q$.
  function $\textsc{OptRel}(Q)$
    $D \leftarrow \{\textsc{Dirty}(q) \mid q \in Q\}$
    return $\{\text{argmin}_{q \in Q \wedge \textsc{Dirty}(q) = d} \textsc{Cost}(q) \mid d \in D\}$

Require: $t$ is a table and $\phi$ is a filter predicate.
Ensure: Returns a set of optimal query plans for scanning and filtering $t$, with
  distinct dirty sets.
  function $\textsc{OptFilter}(t, \phi)$
    return $\textsc{OptRel}(\{\sigma_\phi(q) | q \in \textsc{AddImpute}(t, \textsc{Attrs}(\phi))\})$

Require: $Q$ is a map from tables and dirty sets to optimal query plans (possibly
  with imputations), and $\Psi$ relates query plans with join predicates.
Ensure: Returns a map from dirty sets to optimal query plans involving all
  necessary joins
  function $\textsc{OptJoin}(Q, \Psi)$
    for $size \in 1...|\Psi|$ do
      $S \leftarrow subset(\Psi, size)$
      for $\psi \in S$ do
        $S' \leftarrow S \setminus \psi$
        if $joins(S', \psi)$ then
          $S'_{plans} \leftarrow Q(\textsc{Rels}(S'))$
          $t \leftarrow \textsc{Rels}(\psi) \setminus \textsc{Rels}(S')$
          $t_{plans} \leftarrow Q(t)$
          for $l, r \in S'_{plans} \times t_{plans}$ do
            $P_l \leftarrow \textsc{AddImpute}(l, \textsc{Attrs}(\psi))$
            $P_r \leftarrow \textsc{AddImpute}(r, \textsc{Attrs}(\psi))$
            $\textsc{Update}(Q, \textsc{OptRel}(\{p_l \bowtie_\psi p_r | p_l \in P_l, p_r \in P_r\}))$
    return $Q(\textsc{Rels}(\Phi))$
```

Algorithm 2 Top-level query planner with imputations.

```
Require: set of tables $T$, filter predicates $\Phi$, join predicates $\Psi$, a set of projection
    attributes $P$, optional set of grouping attributes $G$ and aggregator function $A$
Ensure: Returns optimized query plan.
  function PLAN($T$,$\Phi$,$\Psi$,$P$,$G$,$A$)
      $Q \leftarrow \{\}$                                        ▷ map from table and dirty set to plans
      for $t \in T$ do
          $\phi \leftarrow$ GETFILTERS($t$,$\Phi$)
          $P_t \leftarrow$ OPTFILTER($t$,$\phi$)
          $Q \leftarrow Q \cup \{(t,\text{DIRTY}(q)) \mapsto q | q \in P_t\}$
      $M \leftarrow \{\}$                   ▷ map from dirty set to plans, with all joins necessary
      $M \leftarrow$ OPTJOIN($Q$,$\Psi$)                                        ▷ optimize joins
      if $G \neq \varnothing \wedge A \neq NULL$ then
          $attrs \leftarrow$ ATTRS($G$)$\cup$ATTRS($A$)
          $M \leftarrow$ OPTREL($\bigcup_{q \in M}$GROUPBY(ADDIMPUTE($q$,$attrs$),$G$,$A$))
      else
          $M \leftarrow$ OPTREL($\bigcup_{q \in M} \pi_P$(ADDIMPUTE($q$,$P$)))
       return argmin$_{q \in M} Cost(q)$
```

Algorithm 3 An algorithm for in-plan histogram updates

```
Require: $H$ is a map from attribute names to histograms, $op$ is an operator node
    in a logical query plan
Ensure: Returns an updated histogram, such that the distribution of data
    matches the original $H$
  function UPDATEHISTOGRAM($H$,$op$)
      if $op \notin \{\delta,\mu,\sigma,\bowtie\}$ then return H
      $H' \leftarrow$ COPY($H$)
      if op $= \delta_C$ then
          for $c \in C$ do
              $H'[c][null] \leftarrow 0$
      else if op $= \mu_C$ then
          for $c \in C$ do
              ADD($H'[c]$,$H'[c][null]$)    ▷ Adds to histogram based on distribution
      else if op $= \sigma$ then
          SCALEBY($H'$,SELECTIVITY($\sigma$))  ▷ Scales all buckets by constant factor
      else if op $= \bowtie_\phi$ then
          SCALETO($H'$,CARDINALITY($\bowtie_\phi$))                                ▷
  Scales all buckets to sum to constant
      return $H'$
```

technique of *multiple imputation* can be used, in which multiple distinct copies of the complete data are generated and estimators are computed by averaging over the multiple datasets. In ImputeDB, we cannot assume the user desires multiple copies of the query result, though we note that in further work, multiple imputation could be used within ImputeDB to reduce the variance of estimates of some aggregates, like averages.

Our algorithm proceeds by iteratively fitting regression trees to a subset of the data and the target imputation columns. In each iteration, the missing values of a column are replaced with newly imputed values. With each epoch, the quality of the imputation improves as values progressively reflect more accurate relationships amongst 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. Algorithm 4 provides details on the implementation.

The flexibility of this non-parametric imputation approach aligns well with the spirit of query optimization, which aims to reconcile performance with a declarative interface to data manipulation. By removing concerns for imputation, users executing queries in ImputeDB don't need to consider the potential nature of missing data.

It is worth noting that given the nature of the imputation strategy used here, imputation operators become blocking in our system. An extension of ImputeDB could consider regression algorithms that learn in an online nature; this would reduce the cost of the imputation at the possible expense of quality. Again, given the imputation strategy-agnostic design of ImputeDB, this would not pose a problem.

Algorithm 4 An algorithm for chained imputation using regression trees

```
Require: $T$ is a table. $D$ is a set of attributes of $T$ that need to be imputed, and
  $C$ is a set of attributes of $T$ that have complete data
Ensure: Returns an imputed $T$
  function IMPUTEWITHCE-CART($T$,$D$,$C$)
    $T' \leftarrow$ IMPUTERANDOM($T$,$D$)
    for $1...EPOCHS$ do
      for $d \in D$ do
        $imp \leftarrow$ TRAINRT($\pi_{C \cup D \setminus \{d\}}(T')$,$\pi_d T'$)
        $T' \leftarrow (\pi_{C \cup D \setminus \{d\}}(T')$,PREDICTRT($imp$,$T$))
    return $T'$
```

4.7 Complexity

Our optimization algorithm builds off the approach taken by System R [8], therefore our algorithm still operates in exponential time. Indeed, note that if we remove our restriction on types of imputation(δ_{min},μ_{min},μ_{max}), and allow any arbitrary subset of attributes to be imputed, then every single operator in the query plan has a number of imputations exponential in the number of dirty columns. Our restriction, instead, increases the number of plans (in the worst case) at each operator by a factor of 3. Of course, this implies that in the worst case (where all dirty sets tracked are distinct throughout the query plan), we explore a number of plans that further scales the number of plans in the original algorithm by an exponential factor.

However, we note that in all cases we have considered, this exponential blowup does not affect the practical performance of our optimizer.

5 Experiments

To evaluate the performance of our system, we generate plans for queries on two data sets: the American Community Survey and a synthetic dataset.

5.1 Data sets

The American Community Survey (ACS) provides a number of public data sets collected by the U.S. Census Bureau. We used a cleaned version of the 2012 Public Use Microdata Sample (PUMS) data kindly provided by the authors of [3]. The cleaning procedure is described in detail in [3], but to summarize, the following data were removed:

- Rows corresponding to vacant houses or single occupant households.
- Identification variables (e.g. area codes, serial numbers).
- Flag variables.
- Continuous variables.

The final data set consists of a single table with 671,153 rows and 37 columns, where all column entries are integers.

In order to evaluate our implementation, we created a dirtied variant of the ACS data by randomly eliminating 10% of the fields across the table.

In order to evaluate additional queries, we constructed a synthetic table, consisting of 10,000 rows and 10 columns, with all values drawn from a uniform distribution between 0 and 100. To create a dirtied variant of the synthetic data, we randomly deleted 30% of the fields across the table.

5.2 Queries

We collected a set of queries (Table 1) that we think are both representative, in that they could reasonably be written by a user in the course of data analysis, and interesting to plan.

The queries on the ACS data all consist only of projections, selections, and aggregations. The ACS data is contained in a single table, so there are relatively few join queries that are interesting on this data set and remain relevant for analysts, rather than contrived solely for the purpose of experimentation. However, even with select, filter, and aggregate, there are interesting choices to make with imputation placement. In order to explore joins, we leverage the synthetic dataset and write various ad-hoc queries.

#	Query
1	`SELECT BLD FROM acs;`
2	`SELECT BLD as units_in_structure, COUNT(ST) as estimate FROM acs GROUP BY BLD;`
3	`SELECT BATH as has_bath, COUNT(ST) as ct FROM acs GROUP BY BATH;`
4	`SELECT ACR as lotsize, AVG(BDSP) as avg_num_bedrooms FROM acs GROUP BY ACR;`
5	`SELECT PSF as has_sub_families, SUM(NP) as num_people FROM acs GROUP BY PSF;`
6	`SELECT AVG(NP) as avg_num_people FROM acs;`
7	`SELECT BDSP as num_bedrooms, AVG(ACR) as avg_lot_size FROM acs WHERE VEH >= 2 GROUP BY BDSP;`
8	`SELECT MIN(RMSP) as min_num_rooms FROM acs WHERE RWAT=2;`
9	`SELECT * FROM dirty m where m.f1 >= 2;`
10	`SELECT * FROM dirty where dirty.f1 <= 2;`
11	`SELECT MAX(f2) as max_f2 FROM dirty GROUP BY dirty.f1;`
12	`SELECT m1.f1, m1.f2 FROM dirty m1, clean m2 WHERE m1.f1 = m2.f1 and m1.f1 = 1 and m2.f2 = 69;`
13	`SELECT d1.f1 FROM dirty d1, clean d2 WHERE d1.f1 = d2.f1 and d2.f2 = 50;`

Table 1: Queries used in our experiments.

Table 2 provides a summary of our experimental results. We run each query in four configurations: without imputation on clean data, without imputation on dirty data, and with imputation on dirty data for $\alpha = 0$ and $\alpha = 1$. When a query returns a relation of the same "shape" (same number of tuples and same schema) on both clean and dirty data, we can compute an error value. For our experiments, we report root-mean-square error (RMSE) for the base error (i.e. the error in the dirty data without imputation with respect to the clean data) and the change in RMSE for the imputed plans. For imputations, we run queries using the two extreme values of α: 0 and 1, representing a focus on runtime and on information loss, respectively. For each query, the table displays the base error, the change in error in both imputation scenarios, and the running time in seconds.

Planning times are not shown, as they are fractions of a second for all of the queries that we tested. The overall low planning times highlight the practicality of our algorithm, despite the exponential complexity. Running times clearly vary based on α, allowing users to tune their query performance as desired.

We can group our queries into five categories: sum/count queries, average queries, min/max queries, selection queries, and join queries on synthetic data.

- For sum and count queries (2, 3, 5), imputation does a good job of reducing the error. For each of these queries, we can reduce total error by several orders of magnitude. Imputation performs very well in these cases, even for data which is missing uniformly at random, because each missing value directly increases the error of the result.
- For average queries (4, 6, 7), imputation does not reduce the error, mainly because running these queries on dirty data incurs only a small penalty. This is because, in the case of average, removing data uniformly at random does not change the mean. If we were to remove data in a biased way, imputation would

		Imputed ($\alpha = 0.0$)		Imputed ($\alpha = 1.0$)	
#	Base error	Error %Δ	Time (s)	Error %Δ	Time (s)
1	–	–	1	–	17
2	1.66×10^{4}	0	1	-69	9
3	4.78×10^{4}	0	3	-100	9
4	1.86×10^{-3}	0	2	2719	32
5	2.52×10^{5}	0	1	-87	25
6	9.89×10^{-4}	0	1	10003	8
7	3.76×10^{-2}	0	2	112	55
8	0.00	0	2	0	3
9	–	–	0	–	7
10	–	–	0	–	0
11	0.00	0	0	0	0
12	–	–	0	–	0
13	0.00	0	0	0	72

Table 2: Base error, percent change in error and and running time for queries with different imputation levels. Base error is the root-mean-square error (RMSE) between the query run on clean data and the query run on dirty data without imputation. Change in error is relative to the base error.

likely perform better because there would be more error for it to correct.

- For a min query (8), it is unlikely that imputation will ever help, because by definition these queries are looking for extreme values. These values are unlikely to be the result of imputation and are also unlikely to be randomly removed from the dataset.
- We do not compute error for selection queries (1, 9, 10, 12), because the output of the query on clean data is not directly comparable to the output on dirty data.
- Imputation performs poorly on the join queries on synthetic data (11, 13) because the data is synthetic, so there are no correlations for imputation to find. We included these queries because we wanted to test the performance of our algorithm on queries that include joins, and we were pleased to find that simple joins cause no perceptible performance problems.

Finally, we performed CE-CART imputation on the entirety of the ACS table, in order to compare our system to the traditional, pre-processing approach to imputation. We were unable to complete the full imputation due to resource constraints, but the process ran for more than one hour without terminating. This time compares favorably to our maximum of 72 seconds for a query with imputation on the ACS data. Given the varying needs of database users, ImputeDB clearly allows flexibility without a significant performance hit.

6 Conclusion

As shown in ImputeDB, missing values and their imputation can successfully be integrated into the relational calculus and existing 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, but effective, cost model to consider tradeoffs in information loss and time. In effect, user preferences for one or the other can be easily incorporated by adjusting a parameter. Simple histogram transformations provide incrementally updated cardinality estimates across operators in the plans, which allow us to provide more accurate cost estimates. By taking a dynamic programming approach, we can consider a variety of operator placements and input columns, while keeping planning tractable in real-world examples.

In our experiments, we considered a series of analytic queries (using relevant aggregates) and set queries (with simple projections) on real-world American Community Survey data and synthetic data and showed that the chosen query plan improves accuracy of results relative to simply ignoring missing values. Furthermore, the variety of imputation operator placements across queries further emphasizes the lack of flexibility imposed by the coarse-grained pre-processing approach to imputation. By allowing different imputation strategies for different queries, we take a fine-grained approach to missing data, better addressing the often differing needs of database users.

We highlight the long history of dealing with missing data both in the statistical learning and database communities. Similarly 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 the statistical learning work, our emphasis is not on the specific algorithm used to impute but rather on the timing of imputation 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 and engage in normal workloads.

6.1 Future Work

ImputeDB opens up multiple avenues for further work. For instance, we could extend our minimal/maximal imputation operators to consider global information, such as the specific columns needed in operators higher up in the query plan. Our optimizer uses the same machine learning algorithm in all instances of imputation operators. Integrating multiple possible algorithms could allow for further fine-grained imputation, and honing the time complexity and loss of these algorithms for an iterator model database could facilitate more intelligent query plans and consistent interpretations of α across strategies. For example, algorithms that learn in an online manner could increase the efficiency of the system. Furthermore, a multiple imputation strategy could be followed for queries involving certain aggregates. Finally, the underlying database for ImputeDB is far from a full-featured, production database, so a natural step is to integrate imputation planning in a widely-adopted production-quality database. This will likely expose further opportunities for improvement.

References

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

[2] L. F. Burgette and J. P. Reiter, "Multiple imputation for missing data via sequential regression trees", *American Journal of Epidemiology*, kwq260, 2010.

[3] O. Akande, F. Li, and J. Reiter, "An empirical comparison of multiple imputation methods for categorical data", *arXiv preprint arXiv:1508.05918*, 2015.

[4] E. F. Codd, "Understanding relations", *ACM SIGMOD Record*, vol. 5, no. 1, pp. 63–64, 1973.

[5] J. Grant, "Null values in a relational data base", *Information Processing Letters*, vol. 6, no. 5, pp. 156–157, 1977.

[6] J. M. Hellerstein, "Quantitative data cleaning for large databases", 2008.

[7] V. Mansinghka, R. Tibbetts, J. Baxter, P. Shafto, and B. Eaves, "Bayesdb: a probabilistic programming system for querying the probable implications of data", *arXiv preprint arXiv:1512.05006*, 2015.

[8] M. W. Blasgen, M. M. Astrahan, D. D. Chamberlin, J. Gray, W. King, B. G. Lindsay, R. A. Lorie, J. W. Mehl, T. G. Price, G. R. Putzolu, *et al.*, "System r: an architectural overview", *IBM systems journal*, vol. 20, no. 1, pp. 41–62, 1981.

[9] C. Zukin, "What's the matter with polling", *New York Times*, 2015.

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

[11] S. van Buuren and K. Groothuis-Oudshoorn, "Mice: multivariate imputation by chained equations in r", *Journal of Statistical Software*, 2011.