ZIP: Lazy Imputation during Query Processing

Yiming Lin
University of California, Irvine
yiminl18@uci.edu

Sharad Mehrotra
University of California, Irvine
sharad@ics.uci.edu

ABSTRACT

This paper develops a query-time missing value imputation framework, entitled ZIP, that modifies relational operators to be imputation-aware in order to minimize the joint cost of imputing and query processing. The modified operators use a cost-based decision function to determine whether to invoke imputation or to defer to downstream operators to resolve missing values. The modified query processing logic ensures results with deferred imputations are identical to those produced if all missing values were imputed first. ZIP includes a novel outer-join based approach to preserve missing values during execution, and a bloom filter based index to optimize the space and running overhead. Extensive experiments on both real and synthetic data sets demonstrate 10 to 25 times improvement when augmenting the state-of-the-art technology, ImputeDB, with ZIP-based deferred imputation. ZIP also outperforms the offline approach by up to 19607 times in a real data set.

PVLDB Reference Format:
Yiming Lin and Sharad Mehrotra. ZIP: Lazy Imputation during Query Processing. PVLDB, 17(1): 28 - 40, 2023.
doi:10.14778/3617838.3617841

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/yiminl18/QDMIDB.git.

1 INTRODUCTION

A large number of real-world datasets contain missing values. Reasons include human/machine errors in data entry, unmatched columns in data integration [32], etc. Failure to clean the missing data may result in the poor quality of answers to queries that may, in turn, negatively influence tasks such as machine learning [34], data analytics, summarization [25, 27], etc. built on top of data.

Missing value imputation has been extensively studied in the literature, especially from the perspective of ensuring accuracy [16, 36, 43, 47]. Traditionally data cleaning (including missing value imputation) is performed as a data preparation step prior to analysis in data warehouses. Such an offline cleaning approach [31, 42] can, however, become prohibitively costly if the volume of data is large and cost per imputation is high. Data cleaning/imputation is sometimes performed on dirty data as it arrives during ingestion in an online manner [29]. Such ingestion time imputation approaches, however, also becomes impractical if rate of data arrival exceeds the rate at which it can be cleaned. Consider a use case scenario which motivates our work. At UC Irvine, for the past 4 years, we have been using continuously generated WiFi connectivity data captured over a campus wireless networks for fine-grained localization using a Wifi-based localization framework entitled LocatER[37]. LocatER exploits a person's recent connection history to predict the room a person is in given the access point a person is connected to (which corresponds to an imputation problem [37]). LocatER takes roughly 400ms per event for such an imputation. [1] With 1000s of WiFi access points, about 30,000+ individuals connected to the network, and tens of thousand of WiFi events per second, it would take over an 1 hour of processing per one second of data collected from the WiFi infrastructure during peak load. An online approach that imputes location value as soon as data is ingested is clearly infeasible. Likewise, collecting and processing raw WiFi data periodically using LocatER as an offline approach discussed earlier is also equally impractical. Instead, we adopt an alternate query-time approach that cleans data lazily when the need arises. Motivated by similar requirements as the example above, a query-time approach to cleaning/imputation has become popular in several recent studies [12, 13, 17, 21, 23, 24] discussed in related work (Section 2).

Query time cleaning offers several benefits. It significantly reduces the wasted effort and computational resources by cleaning only parts of the data actually needed in analysis instead of indiscriminately cleaning the entire dataset. This is especially important when cleaning is expensive and/or datasets are very large, making cleaning of the data fully infeasible. Predicting the dataset analysts might use apriori so as to clean such data as a pre-analysis step is often not feasible (e.g., when a common analysis operation consists of adhoc queries on the data) [14]. In such situations, the only recourse is to support data cleaning with query processing.

Query-time data cleaning opens new challenges, the prominent of which is to minimize cleaning performed during query processing to reduce latency. This paper develops *ZIP*, a la*Z*y *I*mputation query *P*rocessing approach that exploits query semantics to reduce the cleaning overhead. When processing records with missing values, ZIP may delay imputations until later - such a lazy approach to imputing can be beneficial if the record with the missing value get eliminated in the query tree, thus, avoiding imputatons unnecessary for answering the query. Delaying imputations, comes at a increase in processing cost, if imputation could not be avoided. ZIP, given a query plan for an SQL query, develops an execution strategy that minimizes the overall (joint) cost of imputing missing data and executing the query. We illustrate the key intuition behind ZIP through an example below.

1.1 A Case for Lazy Imputation

Consider a real camera-based localization application in Donald Bren Hall building, UCI, which is instrumented with camera used to locate people. A tuple in Camera-Snapshots (Table 1) stores

[1] This is based on a 16 core 2.50GHz Intel Xeon CPU, 64GB RAM, and 1TB SSD.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 17, No. 1 ISSN 2150-8097.
doi:10.14778/3617838.3617841

Table 1: Camera-Snapshots (C)

sid	faceID	Time	location
1	20	12pm	2206
2	41	2pm	NULL (N_1 = 3001)
3	20	1pm	NULL (N_2 = 2206)
4	35	3pm	NULL (N_3 = 2099)
5	NULL (N_4 = 26)	1pm	3119
6	NULL (N_5 = 55)	2pm	2214

Table 2: User (U)

Name	Type	faceID
Mike	faculty	NULL (N_6 =20)
Robert	graduate	65
John	faculty	NULL (N_7 =55)

Table 3: Space (S)

Room	Building
2214	NULL (N_8 =DBH)
2206	DBH
2011	DBH
3119	NULL (N_9 =ICS)
2065	NULL (N_{10} =DBH)

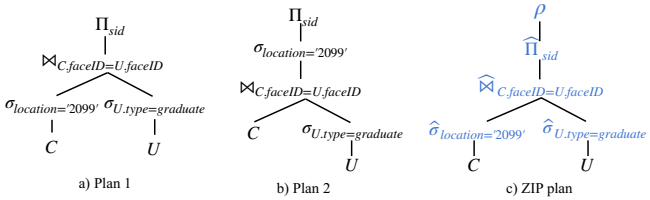


Figure 1: Imputation in Different Query Plans.

the location (i.e., room) of a person (i.e., faceID determined using face recognition) at a given time (i.e., the timestamp). The faceID of a person could be determined by matching camera data with picture(s) of a person stored in the database or through a model trained using such pictures. User (Table 2) and Space (Table 3) tables store the metadata about registered users and space. There are 10 missing values (shown as NULL) in the 3 tables, and we also display the corresponding imputed values (shown as blue color in the bracket). Let us consider a simple query, find all snapshots (sid) for graduate students in room 2099. Such a query joins Camera-Snapshots with the User table, after selecting tuples matching query predicates on each table as shown in Fig 1.

Let us consider various possible query-time imputation strategies in different query plans. Fig 1-a) is the plan where all selections are pushed down. In such a plan, all the missing values under *location* column (i.e., N_1, N_2, N_3) must be imputed since the selection operator $\sigma_{location='2099'}$ require missing values to be imputed prior to execution. After imputations, only one tuple with sid 4 satisfies the selection condition, and will thus be passed onto the join operator. Since the faceID of this tuple (i.e., 35) does not match any faceID in Table 2, the query execution will terminate.

One may be tempted to consider the additional imputation overhead (i.e., N_1, N_2, N_3) of Plan 1 to be a result of pushing selections to the leaf level. This raises an issue whether the savings resulting from modifying the operators could be achieved simply by making the optimizer aware of expensive nature of imputations which may, then, consider imputation required by the selection operator $\sigma_{location='2099'}$ as expensive (as in [26]) resulting in the operator to be pulled above the join condition, such as the Plan 2 in Fig 1-b). Even such a plan would still require 2 imputations for N_4, N_5. Furthermore, such a plan would incur significant execution overhead for tuples for which attribute values are not missing, since the input size to join from table C will be the cardinality of C table without any filtering. Thus, the benefits that can be achieved by modifying the operator implementation cannot simply be mimicked by changing the optimizer.

Now let us now consider the strategy illustrated in Fig 1-c) wherein each operator o is replaced by a corresponding "imputation aware" operator $\widehat{o}$. A modified operator $\widehat{o}$ behaves exactly the same as the original operator o for tuples that do not contain missing values. For instance, for tuples with sid 1, 5, 6 for which location attribute is not missing, $\widehat{\sigma}_{location='2099'}$ evaluates the predicate right away (and drops the tuples since they do not match the predicate). For tuples with missing values (i.e., tuples with sid 2, 3, and 4), the modified operator $\widehat{\sigma}_{location='2099'}$ may decide to either impute the missing value and compute the predicate, or delay the imputation for the downstream operator to perform. Delaying imputation can prevent unnecessary imputations, if such a tuple (whose imputations are delayed) does not satisfy predicates associated with the downstream operators. In our example, if $\widehat{\sigma}_{location='2099'}$ forwards the tuples with sid 2, 3, 4 in Table 1 to the downstream join without imputing N_1, N_2, N_3, it would have resulted in the savings of all the three imputations since the tuples do not meet the join condition (the only graduate student in the User table has a faceID of 65 which does not match the faceID of tuples with sid 2,3, and 4)!

Such a lazy strategy would possibly minimize the imputation costs without sacrificing quality of result. In the example above, saving two or three imputations may appear to be of little benefit compared to the additional complexities that could arise in maintaining state and modifying operators, in practice, when tables are large and imputation costs are relatively expensive such savings quickly add up. For instance, even for the simple query discussed above if Camera-Snapshots contains millions of rows, imputing all the missing locations would be very expensive.

1.2 Challenges in Supporting Laziness

First, while supporting laziness, it is not trivial to ensure the correctness of query answers returned by ZIP, i.e., the query answers returned by ZIP is same as the approach that first imputes all missing values on the entire datasets and then run query processing. To this end, we carefully design the mechanism of the "imputation-aware" modified operator to delay missing values based on outer join strategy and reconstruct the query answers correctly using the replay algorithm. Second, how to adaptively co-optimize the imputation cost and query processing overhead remains a challenging task. In ZIP, we design the decision function to automatically make the decision on whether to delay imputation or impute missing values right away in each operator based on the estimated expectation cost of imputation and query processing. Third, it is critical and non-trivial to perform the laziness in ZIP in an efficient way. We developed techniques to improve the efficiency of ZIP. We exploit the opportunities to remove redundant imputations by leveraging the upstream and downstream predicates using the filter and verify step in the modified operator. What's more, the bloom filter is used to ensure the replay algorithm can be performed efficiently.

1.3 Contributions

The paper introduces a ZIP framework to answer SQL queries over data that may contain missing values. The primary contributions

include (a) simple modification to the logic of relational operators that empowers operators to choose to either impute or delay missing values, (b) a decision-function logic to enable operators to determine whether the imputation should be performed right away or delayed based on a cost-based analysis of tradeoffs between the two choices, (c) efficient mechanisms to maintain state of the execution and the modified query processing logic to continue execution over imputed values so as to generate the right query results. ZIP designs modified operator logic for a wide range of operators, such as selection, join, projection, aggregate-group by, union, set minus and can handle a large class of queries of significant complexity including nested query. ZIP provides orders of magnitude savings - E.g., using ZIP based query processing over ImputeDB query plans can result in savings from 10 to 25 times depending upon the query and data sets. It provides order-of-magnitudes improvement over offline approach (i.e., imputing all missing values in data set and then running query) up to 19607 times in a real data set.

2 RELATED WORK

Missing Value Imputation. As in [17], we view imputation approaches as *blocking* or *non-blocking* in terms of query processing. A blocking strategy reads the whole data to learn a model for imputation (before it imputes any missing value), while a non-blocking strategy can impute missing values independently reading only a (subset of related) tuples. Imputation approaches can roughly be characterized as statistics based, rule based, master data based, time-series based, or learning based approaches [36]. Of these, other than the learning based approaches, many techniques could be used in a non-blocking setting. For instance, ImputeDB [17] used a non-blocking statistics-based mean-value method that replaces a missing value with the mean of the available values in the same column using histograms. Since histograms are often maintained for query optimization and approximate processing [28, 44] such a technique is non-blocking. Strategies that use master data [20, 41, 45, 46] are also non-blocking since they look up a knowledge base and crowd source the imputations one tuple (or a set of tuples) at a time. Imputation strategies in time series data [15, 30, 37] are often performed by learning patterns over *historical* data to forecast *current* missing values or using the correlation across the time series. An example is LOCATER [37] that imputes each missing location of a user at one time stamp by learning user's pattern from historical data. Such methods also clean one tuple at a time and are, hence, non-blocking. Rule based imputation methods based on differential dependency [43] or editing rules [22] often impute missing values by replacing them with corresponding value of similar objects.

In non-blocking strategies the overall cost of imputation is proportional to the number of tuples imputed and hence, ZIP, which is designed to exploit query semantics to reduce number of imputations performed, can bring significant improvements.

In contrast to the above, learning-based approaches [18, 38, 40] are often blocking. ZIP helps learning-based approach by reducing the number of tuples to be imputed (and thus reducing inference time), and ZIP can bring significant improvement when inference time is not negligible, such as KNNImpute [2] which takes 9.73 seconds to impute 1k missing values.

[2]A standard library to impute missing values in scikit-learn in python [8]

ZIP also helps reduce training time since ZIP only learns the model on the columns required by the query. For instance, consider AdventureWork dataset [6] which contains more than 200 columns with missing values. Instead of learning models for all such columns as the offline approach does, ZIP only needs to learn models for only a few columns required by the query. Note that the learned models can be reused and thus save training time for later queries that requires imputations on the same columns.

Additionally, given the significant training overhead of learning based approaches, several prior works have explored reducing the training time of learning-based methods by using sampling [40] or histograms [4]. In this situation, when learning-costs can be brought down to make blocking strategies practical in online settings, ZIP can bring further improvements by reducing redundant imputations.

In summary, ZIP provides a *framework* for lazy imputations during query processing where *any* imputation approach could be properly used in ZIP. ZIP will adaptively adjust its behaviors (impute now or delay imputations) when using imputations with various complexity, and it will provide the most advantages when the imputation cost is significant and data is large. When cheap imputations are used, ZIP will not be any worse compared to the offline approaches since ZIP does not repeatedly clean - it does so only once and then subsequently remembers the imputed values. Note that ZIP does not require an analyst to pre-decide what type of imputation functions to use (cheap, expensive, or in-between) in advance, since ZIP will adaptively adjust its behaviors (impute now or delay imputations) by estimating the imputation cost and query processing cost.

Query-Time Data Cleaning. Query-time strategy has been explored in several data cleaning problems. [24] explores analysis-aware conflicting values detection and repair in database. Specifically, their approach performs repair of denial constraint [19] violations on-demand to integrate data cleaning into the analysis by relaxing query results. QDA [12, 13] develops query-driven approach for entity resolution problem with the goal of reducing the number of cleaning steps that are necessary to exactly answer selection queries. ImputeDB [17] explores a dynamic optimization strategy to design query plans for queries over relations with missing data. In particular, ImputeDB introduces 2 new operators - drop and impute. For any predicate where the condition being evaluated is over an attribute that may contain missing values, ImputeDB introduces one of these two operators. For any tuple that passes through the impute operator, ImputeDB will call the corresponding imputation function to resolve the tuple prior to passing it to the predicate in the original operator in the query tree. In contrast, for a drop operator it will simply drop the tuples whose corresponding attribute contains a missing value. The placement of impute/drop operators explores a trade-off between the accuracy of results and the corresponding overheads, specially when imputations can be expensive and dominate the query evaluation cost. While ImputeDB explores such a trade-off to generate a query plan with drop and impute operators, ZIP explores a complementary *execution* strategy by modifying query processing by changing how tuples with missing values are processed by relational operators in order to reduce the need to impute data. In particular, in ZIP, relational operators may delay imputing missing values in the hope that such

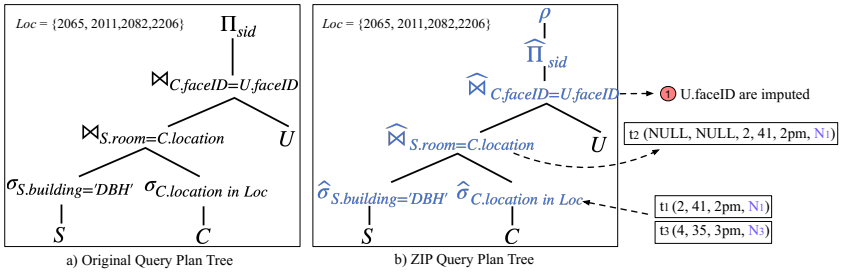


Figure 2: ZIP query plan.

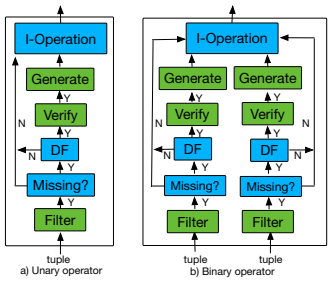


Figure 3: Modified Operators.

```
SELECT C.sid FROM C, U, S
WHERE C.faceID = U.faceID AND
C.location = S.room AND S.building = 'DBH' AND
C.location in {2065, 2011, 2082, 2206}
```

Figure 4: Query

tuples are eliminated by downstream operators preventing the need to impute the missing values. ImputeDB and ZIP can be used in isolation or together since the approaches are complementary. Indeed, in Section 8, we show improvements due to ZIP over query plans already optimized using ImputeDB optimizer to highlight additional advantages that result from ZIP.

3 ZIP OVERVIEW

This section provides an overview of how ZIP achieves delayed imputation by appropriately modifying the relational operators. We will use the query shown in Fig 4 to illustrate ZIP. We shift to this query instead of a simpler query we used which will no longer suffice to illustrate all the cases ZIP needs to handle to ensure correct execution. The Fig 2-a) shows the query tree generated by a third-party optimizer, (e.g., PostgreSQL). ZIP modifies such a plan by replacing operators by their modified versions and by adding a new operator ρ at the top of the tree as shown in Fig 2-b) that imputes missing values whose imputation has been delayed by previous operators. ZIP has been implemented in the context of pipeline query execution using an *Iterator Interface*. The execution starts from the root of query tree by calling a *root.getNext()* that retrieve tuples from the child nodes that satisfy the associated conditions. Child nodes, in turn, recursively call *getNext()* operator on their children. ZIP modifies the relational operators to process incoming tuples that contain missing values differently. Other tuples (that do not contain missing values) are processed exactly as they would be by the original operator. In particular, ZIP does not change the underlying operator implementation - for instance, the relational operator can continue to use hash/sort/nested loop/index-based operator implementations supported in the underlying database without change. ZIP simply routes tuples containing missing values through a sequence of steps (i.e., filter, verify, decision function, generate). Thus, besides code to implement such steps, ZIP only changes the routing logic of operators which requires a very small amount of new code (approx. 500+ lines) while preserving the existing code of the database.

Missing Value Representation: Before we discuss how modified operators are implemented in ZIP, we first specify how ZIP represents missing attribute values. In ZIP missing values are represented using NULLs. However, to differentiate between a value of an attribute being NULL or missing, the relational schema is extended with an additional attribute that contains a bit per attribute of the relation. If the value of attribute a in a tuple t is missing, its value is NULL and its corresponding bit is set to 1. If attribute a is NULL but its bit is 0, then a is not missing, instead it is NULL.

Routing logic of modified operator. Fig 3 shows the modified logic of the unary and binary operators. The tuple incoming to the operator first pass through a *filter* step (the purpose of which will become clear momentarily). It is then checked to determine if the attribute value (on which the operator is defined) is missing (by checking for the appropriate bit in the additional attribute stored in the tuple). If the value is not missing, the tuple is directed to the I-operation, i.e., *Imputation-aware operation*, which, for such tuples, implements the exact same logic as the original operator. If value is missing, the tuple is diverted through a decision function (*DF*) which may decide to either impute or delay imputation of the missing value. If imputation is delayed, the tuple again is routed to the I-operation which, in turn, forwards the tuple to downstream operators with missing value preserved without checking the associated predicate, if any, with the operator. For unary operators (e.g., selection), the tuple is forwarded as is, and for binary operators (e.g., join) the tuples are forwarded to the next operator in the pipeline in a way similar to the way they are in an outer-join as illustrated in the example below. For projection operators, ZIP preserves all attributes in a tuple that contain missing values and that may be imputed later. Tuples that *DF* decides to impute are first routed to *verify* and *generate* steps. The goal of the *verify* step is to determine if the imputed value satisfies all the predicates associated with the previous operators (and hence the tuple would have made it to the current operator). If a tuple t passes the *verify* step, the *generate* step is invoked on t. This step generates all additional tuples that would have resulted by executing the logic of upstream operators had t been imputed at the very beginning of query execution. The generated tuples, now with missing value imputed, are passed through the operator logic and processed just the same way the original unmodified operator would have processed tuples. The imputed value of an attribute a in a tuple t may also be present in other tuples in multi-join queries. When a is imputed, all the tuples with the imputed values will be forwarded to generate step in order to generate all the results as we discussed in Section 5.

We next explain the roles verify, filter, generate and decision function (DF) play in the implementation of the modified operator.

Verify: The verify operator is invoked whenever a missing value is imputed in the current operator to check if, had it been imputed earlier, it would have caused the tuple to be eliminated by a prior upstream operator. In such a case, the tuple can be dropped since such a tuple would not have passed the logic of a prior operator and would, thus, have not reached the current operator.

Filter: Filter operator in ZIP works in a manner dual to verify - while verify is used to check if a tuple whose missing value imputation was delayed in a prior (upstream) operator can be pruned after imputation since it would have failed predicates in prior operators, the filter test is used to prune tuples based on predicates associated with future downstream operators that the tuple will not satisfy. Filter test associated with an operator o can, thus, result in early pruning of tuples saving imputations.

Generate: In operator o, *generate* step is responsible to generate possible tuples that satisfy all the previous upstream predicates of o. If the imputation of N_1 is delayed by the join operator until later (say, until ρ executes), necessary joining tuples that could have resulted from t_1 will need to be generated. To this end, ZIP maintains state of all tuples that flow through the join operator and uses the state to support a carefully designed mechanism that ensures correct query answer even when imputations are delayed.

Decision function: ZIP creates a decision function associated with each operator to determine whether to impute the missing values or delay imputation. Intuitively, it is tempting to delay imputing in operator o if imputations are expensive and the downstream operators of o are selective. If the tuple is eliminated by a downstream operator, imputation required to execute o would be saved. On the other hand, if ZIP decides to impute missing values right away, the imputed tuple will have a chance to be eliminated by the current operator saving execution cost. Decision function is a cost-based solution to estimate the expected execution cost of *imputing* right away versus *delaying* the imputation, and chooses the option with lower cost. We discuss decision function in Section 7.

ρ **Operator**: ZIP adds a new operator ρ at the top of the tree which imputes all missing values in the attributes associated with query predicates that have not been imputed so far. The structure of the ρ operator is same as that of unary operator with the difference that for ρ the *DF* is always set to impute[3]. Like other unary operators, once a tuple is imputed in the ρ operator, it goes through the verify step, and if passing verification, goes through the generate step. Since ρ is the final operator, the way ρ executes the generate differs slightly compared with other operators as will be discussed in Section 6. We note that ρ will impute any missing values in the projected attributes if any and removes all attributes in the imputed tuples that were not part of the projection in the query. [4]

We use a complete example in Figure 2 to illustrate ZIP. Consider a tuple t_1 = (2,41,2pm,N_1) in table C with a missing value in the location field for a query in Fig 2-b. Assume that selection operator delays imputation. Thus, t_1 is passed to the join operator as it is. The modified join operator, which is also defined on the location field, will decide whether to impute the missing location field or to delay its imputation further. If join decides to delay, it preserves the missing value in location in a way similar to the way outer joins preserve tuples. In particular, it generates a tuple t_2 = (NULL,NULL,2,41,2pm,N_1) where N_1 is the preserved missing value and the NULLs represent that the values of those fields are NULL. Here we denote N_i by missing values and the

[3]We could alternatively, also consider drop operator, similar in spirit to ImputeDB, which will allow our technique to explore the cost-quality tradeoff as well.

[4]When a tuple with multiple missing values reaches ρ, ZIP simply prefers imputing attributes in selection conditions. Alternative strategy can be first imputing the missing value with lowest estimated imputation cost.

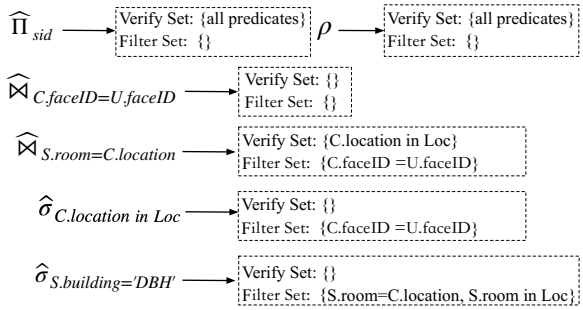


Figure 5: Verify Set and Filter Set.

associated value of *NULL* for null values. Consider the missing value N_1, and assume decision function decides to further delay its imputation in $\widehat{\sigma}_{C.location\ in\ Loc}$. If this value is imputed later during query processing (e.g., in operator ρ), (② in Fig 2), the imputed value must satisfy *every* predicate that applies to the imputed value in the upstream operators prior to ρ. Now consider how filter step works. Assume that the decision function associated with the join operator $\widehat{\bowtie}_{C.faceID=U.faceID}$ decides to impute all missing values of $U.faceID$, and $\widehat{\bowtie}_{C.faceID=U.faceID}$ is implemented as a hash join with U being the *build* table used in the hash join. Then all possible values $U.faceID$ (i.e., $\{20, 65, 55\}$) could take would have been determined early in the pipeline query processing as soon as the build phase of the hash join is complete. Now consider tuple t_3 passing through $\widehat{\sigma}_{C.location\ in\ Loc}$. t_3 will be pruned since its faceID 35 is not in $\{20, 65, 55\}$, and thus not be part of the answer.

Roadmap: In the reminder of the paper, we first describe the implementation of the verify and filter operators in Section 4. The imputation-aware operations (i.e., I-operation) and answer generation (i.e., *generate* step) are described in Section 5 and Section 6 respectively. Finally we show the design of decision function (i.e., *DF* step) in Section 7. We restrict the discussion to the modified versions of the select, project, and join operators and illustrate query processing in ZIP through the SPJ queries. Extensions to other operators other unary and binary operators (aggregation, group-by, union, intersection, set difference) are relatively straightforward and described in the longer version of this paper [11].

4 VERIFY & FILTER STEPS

Data Structures. Implementation of Verify and Filter steps of an operator o requires ZIP to maintain several data structures.

Verify Set. Verify set for operator o consists of all the predicates over the attribute A_o which are associated with all the *upstream* operators (i.e., those that appear below o in the query tree) , where A_o are the attributes associated with the predicate in o. Fig 5 shows the verify sets for all operators in query tree in Fig 2.

Filter Set. A filter set for an operator o consists of predicates defined over attributes associated with the tuples that are input to o. These predicates correspond to conditions associated with operators that are *downstream* to o (i.e., are higher up in the query tree) and are defined over attributes *other than* A_o on which o is defined. As an example, consider selection operator $o = \sigma_{S.Building='DBH'}$ in Fig 2-b) where $A_o = \{S.Building\}$. We add the predicate $\{S.room = C.location\}$ from the downstream join operator to the filter set since it is defined on the attribute $S.room$ which is different from

the attribute *S.Building* on which the selection operator is defined. The filter set for $o = \sigma_{S.Building='DBH'}$ can be expanded further by additional predicates which can be inferred from the current filter set. In the example, the predicate $\{C.location\ in\ Loc\}$ coupled the filter $\{S.room = C.location\}$ enable filter set of $o = \sigma_{S.Building='DBH'}$ to be expanded to $\{S.room = C.location, S.Room\ in\ Loc\}$. The conditions in the filter set are used in ZIP to eliminate tuples earlier thereby saving unnecessary imputations.

Bloom Filters. ZIP constructs a bloom filter [5] for each join attribute in the equi-join operator. Such a bloom filter, $BF(a)$ for the attribute a is constructed incrementally as the tuples are processed by the modified join operator. When the modified operator processes a (non-missing) attribute value, it stores the value into the bloom filter $BF(a)$. Likewise, whenever a missing value in a tuple for a join attribute is imputed (either as part of the join or a further downstream operator) and passes the verification test, it is added into the corresponding bloom filter. The bloom filters help prune/-filter tuples early in upstream operators based on downstream join conditions. For instance, in the example above, using the bloom filter, the operator $\sigma_{S.Building='DBH'}$ could use the join condition in its filter set (e.g., $\{S.room = C.location\}$ to check if the *room* associated with the current tuple matches *any* $C.location$ using the bloom filter $BF(C.location)$. In addition to helping implement the filter step, ZIP uses bloom filter in the modified join will also be used to support the modified join operator (in Section 5).

Bloom Filter Completeness. A bloom filter $BF(a)$ for a join attribute a in a query Q is said to be *complete* with respect to Q if $BF(a)$ *contains all values of a that could result in tuples in the answer set of* Q. Note that the completeness condition does not require all values of a to be in $BF(a)$. Tuples that are filtered away by the selection/join operators (and hence do not contribute to the query answers) may not be in $BF(a)$ for it to be considered complete. Let Q be a query over relations $R_1, R_2, \ldots, R_n$. W.L.O.G, let a be an attribute in R_1 that participates in a join predicate in Q. The bloom filter $BF(a)$ is said to be complete w.r.t. Q if for all tuples $t_1 \in R_1$, such that there exists tuples $t_2, t_3, \ldots, t_n$ in $R_2, R_3, \ldots, R_n$ that along with t_1 produce a tuple in the answer set, $BF(a)$ contains $t.a$.

We denote the event during query processing that causes the bloom filter $BF(a)$ to become complete as $BFC(a)$ and we refer to it as completeness event for $BF(a)$. For $BFC(a)$ to be reached, two conditions should be held. First, all the tuples with missing values in a should have been imputed or eliminated and there is no missing values. To test such a condition, for a query Q, ZIP maintains a missing value counter $MC(a)$ that records the number of missing values for each attribute a in Q. Such an array is initialized using the metadata or statistics maintained by in database. Whenever a missing value in attribute a is imputed or dropped, (e.g., as a result of a filter operator), ZIP reduce the count of $MC(a)$ appropriately.

Second, reaching $BFC(a)$ further depends upon the specific join algorithm used to compute a join. Consider a join $R^L.a \bowtie R^R.b$, where R^L and R^R are the left and right relations respectively, and a and b are join attributes. If no ambiguity, we will refer to R^L and R^R simply as L and R. If $L.a \bowtie R.b$ is implemented using nested loop, for inner relation R, bloom filter $BF(R.b)$ contains all values in $R.b$ (i.e., $BFC(R.b)$ is reached) when there are no more missing values of $R.b$ and the first pass of relation R has been processed. For outer relation L, such a condition becomes true only when all tuples have been processed through the join operator. For hash joins, the bloom filter contains all values as soon as the hash table based on build relation has been built and for outer relation such a condition is reached when all tuples have been processed. For sort merge, or multi-pass hash join, the bloom filters for both relations L and R contains all values when the sort or hash table build is finished. ZIP maintains for each attribute a in a join a boolean, entitled $JC(a)$ that becomes true when all the values in a have been processed. ZIP modifies the scan operator to detect and set JC conditions when all tuples in a relation have been consumed.

Thus to determine $BFC(a)$ ZIP simply needs to check when both $MC(a) = 0$ and $JC(a) = true$ has been reached.

Verify and Filter Implementation. To implement verify and filter operation, for incoming tuples, ZIP only needs to check conditions stored in verify and filter set to determine if the tuples satisfy them or not. If the conditions are selections, tuples can be evaluated right away. For join conditions, we check if the bloom filters of the join attributes are complete or not. If they are complete in pipeline query processing, we use the bloom filter to test if the tuple satisfy this join condition. For instance, consider operator $\widehat{\sigma}_{S.building='DBH'}$ whose filter set contains a join condition $S.room = C.location$ in Fig 5. For tuple t received by $\widehat{\sigma}_{S.building='DBH'}$, if $S.room$ is not missing, and the bloom filter $BF(C.location)$ is complete, ZIP uses $BF(C.location)$ to check if $S.room$ has any matched values in $BF(C.location)$. If $BF(C.location)$ returns false, we drop tuple t. This check operation is safe because bloom filter does not have false negative. Else, we *do nothing* and let t pass.

5 IMPUTE-AWARE OPERATORS

In this section we describe the impute-aware operation, i.e., I-operation in Fig 3, for selection, projection, join and ρ operators.

Unary Operators: I-operation for the select, project and ρ operators are straightforward. For selection, I-operation simply evaluates the selection predicate if the corresponding attribute value is not missing. Else, it forwards the tuple to the next operator. I-operation for the projection operator, besides forwarding attributes in the projection, also preserves values associated with attributes in query predicates for tuples that have missing values in those attributes. The I-operation for the ρ operator at the top of tree returns the tuples after projecting to the attributes in the final results. We illustrate the execution using an example in Fig 6. Fig 6-a) is the ZIP query plan for query in Fig 4, and the decisions taken by the decision functions in each operator are marked. In Fig 6-b) to g), the numbered red circle represents the tuples returned by $getNext()$ for each operator. Assume ZIP decides to delay imputations in two selection operators $\widehat{\sigma}_{S.Building='DBH'}$ and $\widehat{\sigma}_{C.location\ in\ Loc}$, and their getNext() tuples are shown in Fig 6-b) and Fig 6-c), respectively. The projection operator $\widehat{\Pi}_{sid}$ returns tuples in Fig 6-f), it not only projected *sid*, but also all the attributes in query predicates.

Join Operator: The I-operation for the join operator is more complex. Consider modified join operation $\widehat{\bowtie}_{L.a=R.b}$, and a tuple t that reaches I-operation of the join in either relation L or R. Note that such a tuple t has passed through the *filter*, *decision function*, *verify* and *generate* steps in Fig 3. W.L.O.G, let t belongs to relation L. First, if the attribute value a in t, $t.a$, is not missing, ZIP adds

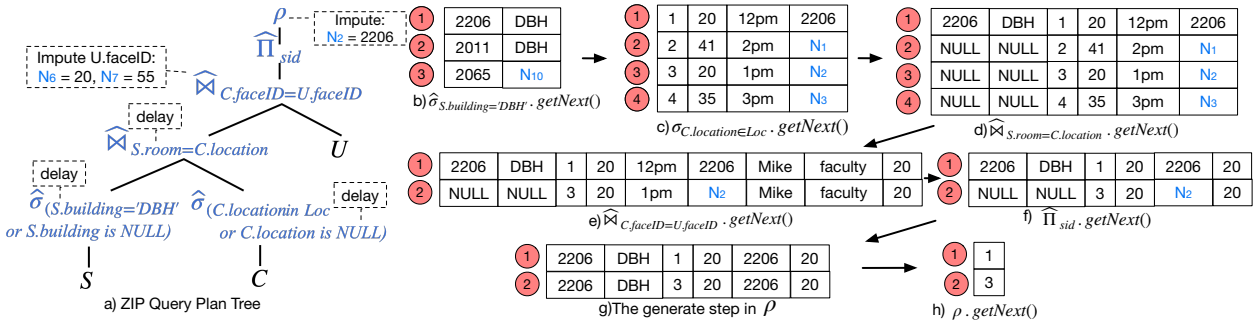


Figure 6: Pipeline ZIP.

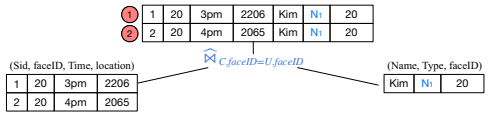


Figure 7: Example of Missing Value Duplication.

$t.a$ into the bloom filter $BF(a)$ and simply uses the original join implementation to join t with tuples in R whose b values are not missing. For instance, if the query plan specified a hash (or a index, or nested-loop) join, ZIP simply continues to use the original code for such joins that were part of the database prior to modifying the operators to be impute-aware. If $t.a$ is missing, however, ZIP bypasses the original join code and instead generates a new output tuple that contains all the attribute values of t including the missing value, and NULL values for all the attributes of the other relation (i.e., ZIP preserves the missing value of $t.a$ for later query processing by creating a tuple similar to the tuple created by the left-outer join). Likewise, if $t \in R$, then ZIP creates a corresponding tuple by concatenating NULLs for the attributes in L (i.e., creating a tuple as would be created by the right-outer join).

In addition, for one of the two inputs (i.e., L or R) for the join operator $o = \widehat{\bowtie}_{L.a=R.b}$, ZIP maintains a list of tuple identifiers of the base relations from which the missing value of $L.a$ or $R.b$ originated. These lists are denoted by $\mathcal{L}(o, a)$ and $\mathcal{L}(o, b)$ respectively. ZIP only populates one of $\mathcal{L}(o, a)$ and $\mathcal{L}(o, b)$ leaving the other empty. ZIP chooses the list that is expected to be smaller (e.g., with lesser number of missing values in the corresponding base relation) to reduce overhead. Thus, if either of the two inputs do not contain missing values, ZIP will choose that attribute, and hence both lists would be empty. These lists, as we will see in Section 6, are required to ensure result tuples are generated only once with no duplicates.

To illustrate the modified join operator, consider join operator $\widehat{\bowtie}_{S.room=C.location}$ in Fig 6-d), where only $C.location$ has missing values. We assume decision function decides to delay imputation in this join operator. The tuple ① in Fig 6-d), is a joined tuple from tuple ① in S relation in Fig 6-b) and tuple ① in C relation in Fig 6-c). All the other tuples, i.e., tuples ②-④ in Fig 6-e), are the right outer join results of S and C where the missing values N_1, N_2, N_3 are preserved with NULLs in columns in S side.

Note that a missing value may appear in multiple tuples if one tuple t matches multiple tuples in join operation. In Fig 7, in the join operation $\widehat{\bowtie}_{C.faceID=U.faceID}$, N_1 appears more than once in the join result. To prevent having to impute the same missing value more than once, when a missing value is imputed, references to the value in all tuples are replaced at the same time. For this purpose, we maintain a link from the missing values to all tuples in which they appear. When a missing value is imputed, all of these tuples will (with the missing attribute imputed) will be passed on to the generate step to compute the corresponding results.

6 THE GENERATE STEP

This section describes how ZIP generates tuples when a tuple with an attribute (whose imputation had been delayed by a previous operator) is imputed as part of a downstream operator (e.g., another relational operator or the ρ operator). Let *generate* step be invoked when a missing value in attribute a of a tuple t is imputed and passes the verify step for an operator o. Generate reconstructs all the tuples that would be present in the output of o to which $t.a$ would have contributed, had $t.a$ been imputed earlier. The essential idea in the generate step is to *replay* the joins on tuple t contributed by the imputation of $t.a$, as shown in Algorithm 1.

Given operator o, let $UJ(o, a)$ be the set of predicates associated with join operators upstream of o whose associated predicate contains attribute a. $UJ(o, a)$ can be identified from the join predicates in the verify set of o. For instance, $UJ(\rho, C.location) = \{S.room = C.location\}$. Note that $UJ(\rho, C.location)$ does not contain join condition $C.faceID = U.faceID$ even though it is also an upstream join of ρ since it does not contains $C.location$. If $UJ(o, a)$ is empty, then tuple t is forwarded to the I-operation in o. (Ln.8) Else, for each predicate p in $UJ(o, a)$, the generate step first checks if the attribute (other than a) in p has reached its bloom filter completeness (i.e., $BFC(b)$ is true, where b is an attribute in p, and $b \neq a$) by calling CHECK_REPLAY_READY_(o, a). In such a case it generates all the tuples that would have resulted from the imputed value of a in t by replaying the joins (Ln.3-7). Note that if any attribute present in predicates in $UJ(o, a)$ (other than a) is not bloom filter complete, the join processing for the tuple containing the imputed value cannot be processed fully right now. Hence, the original generate step after imputation would simply forward the tuple to the I-operation of o which will forward it to downstream operators for future processing similar to the way I-operators pass tuples containing missing values (Ln.8). If o is the ρ operator (and, thus, there is no further downstream operator for o to push the tuple whose join processing is not complete), ZIP banks such tuples whose CHECK_REPLAY_READY_(o, a) fails until the time the condition becomes true. Once the condition becomes true, the tuple is rerouted to generate all the relevant results using the replay function. For tuples not delayed by ρ, all the generated answers are returned as output by passing the tuples to the corresponding I-operation of ρ.

Algorithm 1: *generate* step in operator o

Input: o, t, a
1 $\mathcal{T} \leftarrow \{t\}$
2 **if** CHECK_REPLAY_READY_(o, a) **then**
3 $new\_\mathcal{T} \leftarrow \emptyset$
4 **for** $o_j \in UJ(o, a)$ **do**
5 **for** $t_i \in \mathcal{T}$ **do**
6 $new\_\mathcal{T} \leftarrow new\_\mathcal{T} \cup$ REPLAY(t_i, o_j, a)
7 $\mathcal{T} \leftarrow new\_\mathcal{T}$
8 **return** $\mathcal{T}$ to the I-operation of o

Note that for all tuples delayed in the ρ operator, eventually the CHECK_REPLAY_READY_(o, a) will become true, which requires that the $BFC(b)$ be true for any attribute b that is an attribute in any predicate p in $UJ(o, a)$ other than a. Reaching $BFC(b)$ requires $MC(b) = 0$ and $JC(b) = true$. The condition $JC(b)$, as discussed in Section 4, is a property of the join algorithm used and will eventually always be met for all attributes as the scan for the base relation containing b has processed all the tuples that satisfy the predicates (if any) associated with the scan. The condition $MC(b)$ will also be eventually reached as the ρ operator continues to impute the remaining missing values.

Replay Function: We now explain the REPLAY function in Algorithm 2. Consider executing replay function for a tuple t in operator o, and assume the join condition is $a = b$. ZIP first checks if the imputed value $t.a$ is in the bloom filter of attribute b, i.e., $BF(b)$. If a matched value is not found, then the tuple t will not join with any tuple in current join operator o and thus an empty set is returned. (Ln.2-3) Else, if the bloom filter matches the imputed value of a, ZIP first retrieves all the tuples in the relation that match with t on the join attribute a using the index built on a (Ln.4), and remove the tuples stored in the $\mathcal{L}(o, b)$ to prevent from generating possible duplicated join answers. Its correctness will be clear in later discussion part in this section. ZIP then updates each such matched tuple to t and returns the results by using the merge function. (Ln.7-8) [5]

As an example in Fig 6. consider the generate step in ρ operator in Fig 6-g. When the input tuple to ρ (tuple ② in Fig 6-f), $t = \{NULL, NULL, 3, 20, N_2, 20\}$ with missing C.location N_2 is imputed as 2206, ZIP generates the answers resulting from this imputation for all the join conditions containing C.location in the tree. In this query tree, $\widehat{\bowtie}_{S.Room=C.location}$ is the only upstream join operator of ρ that is applicable to $C.location$. We further assume that the bloom filter of $S.room$ is complete (e.g., $S.room$ does not have missing values and is the build side of join.). When ZIP replays $t = \{NULL, NULL, 3, 20, N_2, 20\}$ using join condition $S.Room = C.location$, ZIP checks $BF(S.room) = \{2206, 2011, 2065\}$ and the imputed value 2206 is found. ZIP then retrieves the matched tuple in relation S which is $\{2206, DBH\}$, and update t to $\{2206, DBH, 3, 20, 2206, 20\}$, as the tuple ② in Fig 6-g. Note that such update can be easily achieved since the schema of each (composite) tuple is maintained in each operator and we could project the matched tuples into corresponding fields in t by aligning their schema.

Discussion: The correctness of ZIP requires 1) soundness: the tuples returned by ZIP would have been returned had we imputed in the base relations prior to executing the query; 2) completeness: ZIP will not miss a result; 3) non-duplicates: ZIP will not generate duplicated results. We focus on joins execution since proving the correctness of other (unary) operators is simpler. Consider join operation $L.a = R.b$, let $\mathcal{T}_d^L$ ($\mathcal{T}_c^L$) be the tuples in L that pass filter and verify steps and have (do not have) missing values in $L.a$. Likewise, $\mathcal{T}_d^R$ and $\mathcal{T}_c^R$ are similarly defined. $L.a \bowtie R.b$ can be rewritten as $(\mathcal{T}_c^L \cup \mathcal{T}_d^L) \bowtie (\mathcal{T}_c^R \cup \mathcal{T}_d^R)$. In join operator $o = \bowtie_{L.a=R.b}$, the I-operation of o will implement $\mathcal{T}_c^L \bowtie \mathcal{T}_c^R$ as normal join. Tuples in $\mathcal{T}_d^L$ and $\mathcal{T}_d^R$ will be pushed to the downstream operators by appending NULLs for the attributes in the other relation. In later query processing, when the bloom filter of $R.b$ is complete, $\mathcal{T}_d^L \bowtie R.b$ will be computed by the generate step. Similarly, when $L.a$ reaches its bloom filter completeness condition, $L.a \bowtie \mathcal{T}_d^R$ will be generated. Note that this may result in the duplicated results for $\mathcal{T}_d^L \bowtie \mathcal{T}_d^R$. Recall that ZIP maintains a list $\mathcal{L}(o, a)$ (or $\mathcal{L}(o, b)$) in every join operator o to prevent the generation of such duplicated tuples.

Algorithm 2: REPLAY

Input: t, o, a
1 b : the join attribute in o other than attribute a
2 **if** $t.a$ *not in* $BF(b)$ **then**
3 **return** $\emptyset$
4 **else**
5 $T_{matched} \leftarrow look\_up(t.a)$
6 $T_{matched} \leftarrow T_{matched} \setminus \mathcal{L}(o, b)$
7 $Ans \leftarrow \emptyset$
8 **for** $t_i \in T_{matched}$ **do**
9 $Ans \leftarrow Ans \cup merge(t, t_i)$
10 **return** Ans

7 DECISION FUNCTION

In the decision function in ZIP, the decision of whether a missing attribute value should be imputed prior to the execution of the operator or should imputation be delayed depends upon whether the imputation method is non-blocking or blocking. We focus on an *adaptive* cost-based solution for non-blocking imputations, denoted by ZIP-adaptive. For blocking imputations such as learning approaches, we use a *lazy* strategy, denoted by ZIP-lazy, which always delays imputing until the tuple with the missing value reaches the imputation operator ρ. When learning based approach is used, ZIP uses the same training data as the offline approach to guarantee that the query answers returned by ZIP is same as the one returned by the offline approach. Although this might increase latency to ZIP if the size of training data is large, there are several practical strategies to mitigate this problem. Once a model is learned, it can be reused to impute missing data in later queries. We thus can employ a warm-up phase to first run a small amount of queries where the models learned for those frequently-queried columns are reused and save learning time for later queries. ZIP-lazy is detailed in the long version of the paper in [11].

Obligated Attributes Non-blocking imputations in ZIP can be placed anywhere in the query tree since ZIP, through operator modification, decouples imputation from the operator implementation. To guide the actions of each operator, we first define a concept of

[5] ZIP requires indices on all join attributes. If such an index does not exist, ZIP will create a hash index as part of the execution of the join operator.

Definition 7.1. **(Obligated Attributes)** Given the set of attributes in predicate set of a query Q (denoted by A_Q), an attribute a in relation R is said to be obligated if

- attribute a appears in a predicate in Q, i.e., $a \in A_Q$, or a is one of the attributes listed in a projection operator; and
- all attributes of R (other than attribute a) do not appear in any predicate in Q. That is, $\forall a^{'} \in R - a, a^{'} \notin A_Q$.

If an attribute $a \in R$ is neither in the projection list nor in A_Q, imputing its missing values will not be required to answer Q and hence a would not be obligated. Likewise, if a predicate in A_Q contains an attribute b which is also in R, it is possible that such a predicate may result in the tuple of R to be eliminated thereby making imputation of the corresponding a value (in case it was missing) unnecessary, which would prevent a from being classified as obligated. U.faceID in Table 2 is a obligated attribute for query Q in Fig 4 because other attributes U.name and U.type are not in any predicate of query Q and U.faceID is in join predicate U.faceID=T.faceID. Since missing values of obligated attributes must always be imputed, there is no benefit in delaying their imputations. In contrast, imputing could potentially reduce number of tuples during query processing. For the remaining attributes, ZIP performs a cost-benefit analysis to decide whether to impute.

Decision function For each operator o in query tree, ZIP associates a decision function $df(a, o)$ for all attribute a that appears in the predicate associated with o. Decision to delay/impute missing values has implications on both imputation and query processing costs. Consider a tuple t in relation $R = (a, b, c, d)$ and a query tree in Fig 8-a). Say $t_1 = (N_1, 1, 2, 3)$ (N represents missing value), if we delay imputing $t_1.a$, and $t_1.b$ does not join with any tuples in the other relation, we can avoid imputing $t_1.a$. On the other hand, imputing $t_2.a$ for $t_2 = (N_1, N_2, 2, 3)$, could prevent imputation of $t_2.b$, if the imputed value of $t_2.a$ is filtered in the selection operator. Imputing $t_2.a$ may also reduce query processing time since it does not require the operator on attribute b to be executed.

Since decisions on whether to impute/delay are made per tuple containing missing values locally by the operator, the decision function must not incur significant overhead. In making a decision for operator o_1 over attribute value $t.a$ of a tuple t, ZIP assumes if t contains other missing values in attributes, say $t.b$ (on which predicates are defined in downstream operators), say o_2, those operators will decide to impute $t.b$ if (and when) the tuple t reaches those operators. For instance, in query tree in Fig 8-a), in making a decision for imputing /delaying $t.a$, i.e., N_1, in developing a cost model we assume that the missing value N_2 ($t.c$) will be imputed right away. This prevents, ZIP to have to recursively consider a larger search space that enumerates (potentially exponential number of other possibilities wherein downstream operators may delay/impute.)

We build a cost model below to estimate impact of delay/impute decision on both the imputation cost and the query processing cost based on which the operators make decisions in ZIP. To compute the imputation and query processing costs associated of the decision for an operator, ZIP maintains the following statistics:

• $impute(a)$: Cost of imputing a missing value of attribute a, computed as an average over all imputations performed so far for missing values of a.

• Selectivity of selection operator o_i, $S_i = \frac{|T_s|}{|T_c|}$, where T_c (T_s) are tuples that are processed (satisfy) the predicate associated with $_i$.

• Selectivity of join operator between relation L and R computed as $S_{o_i} = \frac{|T_s|}{|T_L||T_R|}$, where T_L (T_R) are tuples in relation L (R) and T_s are tuples that satisfy o_i [6]

• $TTJoin_o$: the average time to join tuples in (join) operator o; [7]

• $\mathcal{T}_o$: the average number of evaluation tests to perform per tuple in operator o for tuples without missing values in the attribute to be evaluate in o. [8] If o is join operator, evaluation tests refer to join tests. Else, if o is selection operator, we set $\mathcal{T}_o = 1$.

To bootstrap the process of statistics collection, ZIP initially delays all imputations forcing tuples to rise up to the top of the tree (or be dropped if they fail some predicates en-route). During this process, ZIP collects imputed tuple samples to compute $impute(a)$ and to determine other statistics such as $\mathcal{T}(o)$, join cost $TTJoin_o$ and selectivity S_{o_i}. These statistics are then adaptively updated during query processing.

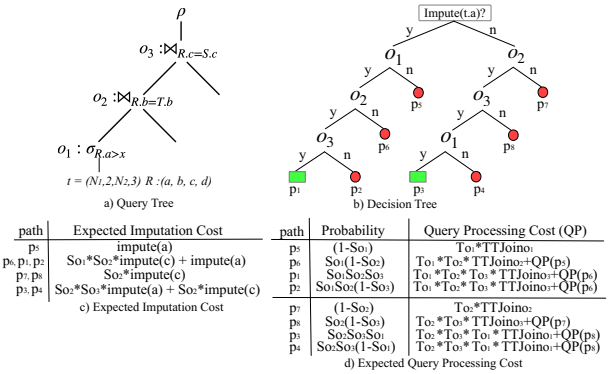


Figure 8: Decision Function Example.

Cost Model for Imputations. We illustrate how to estimate the imputation cost using an example, and include the mathematical model for imputations and query processing in [11].

Consider a query tree in Fig 8-a), and a tuple t =(N_1, 2, N_2, 3). To decide whether to impute or delay missing value $t.a$ (N_1), ZIP estimates the total imputation cost in case it chooses to impute or to delay imputing $t.a$. The set of possible executions that may result for either of the decisions are illustrated in the decision tree shown in Fig 8-b). Each path of the tree corresponds to a possible outcome based on the decision to impute/delay imputing $t.a$. For instance, in path p_5, $t.a$ is imputed but fails the predicate in o_1, while in path p_3, $t.a$ is delayed and t passes the predicates associated with o_2 and o_3, and reaches the imputation operator ρ, where $t.a$ is imputed and evaluated in ρ using predicate associate with o_1. The estimated imputation cost in the case of imputing (delaying) $t.a$, is the summation of the *expected* imputation cost of all paths in the left (right) side of tree, i.e., p_1, p_2, p_5, p_6. (p_3, p_4, p_7, p_8). The expected costs of various paths (shown in Fig 8-c) are computed as a weighted sum of imputations along the path, where the weight corresponds to the probability of execution of that imputation. For instance, for path p_1, we impute $t.a$ with the probability of 1, and, subsequently

[6] We exclude those tuples containing missing values from T_s, T_c, T_L and T_R.

[7] We also use the notation $TTjoin_o$ for selection operator, in this case, $TTjoin_o = 0$.

[8] A tuple with missing value in the attribute that passes through o will be pushed to the above operator without evaluation immediately, and thus $\mathcal{T}_o$ for such tuple is 1.

impute $t.c$ with the probability of $S_{o_1}S_{o_2}$. Thus, the cost of path p_1 is $impute(a) + S_{o_1}S_{o_2}impute(c)$.

Cost Model for Query Processing. Since join costs dominate query execution, ZIP estimates query processing costs by the corresponding join costs. Consider the same decision tree in Fig 8-b). The expected query processing cost if we impute (delay) $t.a$ is the sum of the expected query processing costs for all the paths in the left (right) side of the tree. Fig 8-d) lists the probability of each path, and also, its query processing cost. The probability is estimated based on selectivity of the predicates along the path, and the cost is estimated by summing execution cost of execution of operators along the path incurred in processing tuple(s) that are generated as a result of processing t. Take p_6 as an example. Its corresponding probability is $S_{o_1}(1 - S_{o_2})$ since t passes o_1 but fails o_2. The estimated cost for processing t (shown in Fig 8-a) in operator o_1, denoted by $QP(o_1)$, is $\mathcal{T}_{o_1} * TTJoin_{o_1}$ which is 0 in this example since o_1 is a selection operator for which $\mathcal{T}_o$ is 1 and $TTJoin_{o_1} = 0$. The cost $QP(o_2) = \mathcal{T}_{o_1}\mathcal{T}_{o_2} * TTJoin_{o_2}$ since o_2 is a join operator and $\mathcal{T}_{o_1}\mathcal{T}_{o_2}$ is the estimated number of join tests to perform in o_2. Decision function will decide to impute missing values if the estimated cost of imputation is lower. Otherwise, if the estimated cost of query processing is lower, the imputation will be delayed.

8 EVALUATION

In this section we evaluate ZIP over two real data sets and one synthetic data set. We implemented ZIP on top of a database prototype system, SimpleDB [3] [9]. Note that the ImputeDB optimizer is also implemented in SimpleDB. We did so, so that we can directly measure the improvements due to ZIP on ImputeDB query plans.

8.1 Data & Query sets

WiFi. The first data set consists one week of WiFi connectivity events at Donald Bren Hall Building in UCI. WiFi based occupancy determination has recently received a lot of attention due to the pandemic with several companies offering related products [7, 9, 10] and research projects [33, 35, 37]. The database consists of three tables, *users*, *wifi* and *occupancy* with 4018, 240,065 and 194,172 number of tuples, and totally 383,676 missing values respectively. WiFi records the continuous connectivity data of devices - i.e., which device is at which location in which time interval. *occupancy* records the number of people at different locations over time.

CDC NHANES. We use the subset of 2013–2014 National Health and Nutrition Examination Survey (NHANES) data collected by U.S. Centers for Disease Control and Prevention (CDC) [1]. [10] CDC data set has three tables, demo, exams and labs, which are extracted from a larger complete CDC data set. demo, exams and labs have 10175, 9813, 9813 tuples, respectively, and all of them have 10 attributes. Among them, there are totally 24 attributes that contain missing values, whose missing rate ranges from 0.04% to 97.67%, with total 81,714 missing values.

Smart Campus. Smart-campus data set consisting of 3 synthetic sensor tables, WiFi, Bluetooth, Camera, a space table (that connects sensors to locations) user table (that connects a user to a device mac-address). In addition, two additional tables are derived from the sensor data. The first table location consists of location of user over time and the second table occupancy consists number of people at a given location over time. smart-campus data set has totally 1,892,500 tuples and 1,634,720 missing in occupancy and location and occupancy tables.

Table 4: ZIP VS QTC-Eager.

		Time(secs)			# of Imp ($*10^3$)		
Data Sets	Imputation	QTC-Eager	ZIP-lazy	ZIP-adaptive	QTC-Eager	ZIP-lazy	ZIP-adaptive
WiFi	Mean	1.6	2.3	1.6	79	3.2	74
	LOC	393.4	19.1	19.2	78	3.4	3.6
	KNN	769.2	30.6	-	79	3.3	-
	XGboost	144.7	126.2	-	79	3.2	-
CDC	Mean	0.12	0.12	0.12	12.1	1.3	10.7
	KNN	9.63	1.02	-	12	1.3	-
	XGboost	42.5	37.8	-	11.9	1.3	-
Smart Campus	Mean	3.8	9.6	3.8	16	4.7	15.9
	LOC	97.8	32.5	27.2	16	4.7	4.8
	KNN	157.1	44.3	-	16	4.6	-
	XGboost	101.8	72.6	-	16	4.7	-

Query Set We create three query workloads to evaluate ZIP, *random* (with random selectivity), *low-selectivity* and *high-selectivity*. In each query workload, the majority of queries are SPJ-aggregate queries that contains *select*, *project*, *join*, *aggregate* (*group by*) operations. SP queries are also included. Each query workload contains 20 queries, and we describe the query samples as well as the above three data sets in more detail in the long version of ZIP [11].

8.2 Imputation Methods

Note that any imputation approach could be approapiately used in ZIP, and we choose several popular and easily used methods, which could be called in standard Python library or be well-packaged in Git-hub. For the CDC NHANES dataset we use three imputation approaches, Top-k nearest neighbor [8] (KNN), XGBoost [4, 18], and histogram-based mean value imputation [17]. Of these, the first two are blocking while the third is non-blocking. KNN, XGBoost and mean value imputations are widely used and their implementations are available in standard Python packages, such as sklearn or xgboost. For the WiFi and Smart Campus data set to impute location and occupancy values, in addition to using the above three approaches, we further use a proprietary non-blocking imputation method LOCATER [37] (LOC in short). The complexity of each imputation method is described in the long version of ZIP [11].

8.3 Strategies Compared

We evaluate the two versions of ZIP - *ZIP-lazy* and *ZIP-adaptive* as defined in Section 7 with a baseline query-time strategy, *QTC-Eager*, that imputes missing values eagerly without delay during query execution as soon as the imputed value is required during query processing. Comparing ZIP to QTC-Eager will show benefits of the lazy imputation strategy. We also compare against the offline approach that first imputes all missing data and then run queries. In Experiment 1 to 5 below, we use the query plan generated by PostgreSQL as the input for ZIP to execute. [11] In Experiment 6, we incorporate ZIP with query plans generated from ImputeDB and explore the performance of the combination.

8.4 Results

Experiment 1: ZIP VS Offline. To show the needs of query-time imputation (i.e., ZIP) by comparing with the offline approach, we collected a larger real data set WiFi-large by extending the WiFi data set from one building to 40+ buildings over campus, and we report the runtime of ZIP and the offline approach in Table 5 and Figure 9. In Figure 9, the offline approach takes 6.4 hours in WiFi data set

[9] SimpleDB, developed at MIT has been used for research purposes at several universities including MIT, University of Washington, and Northwestern University.

[10] We thank ImputeDB [17] for providing this data set whose link can be found in [2].

[11] We implement an API in simpleDB that could read and translate the PostgreSQL plan to be executed in its executor.

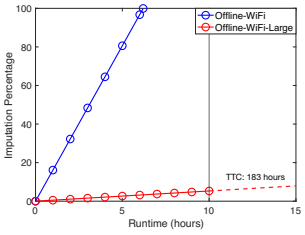
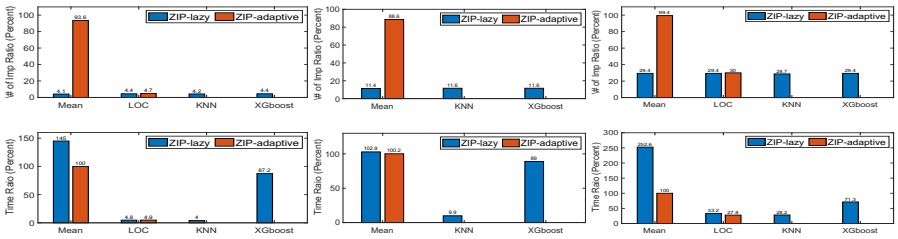


Figure 9: ZIP VS Offline.



(a) WiFi Data. (b) CDC Data. (c) Smart Campus Data.

Figure 10: ZIP VS QTC-Eager.

Table 5: ZIP VS Offline.

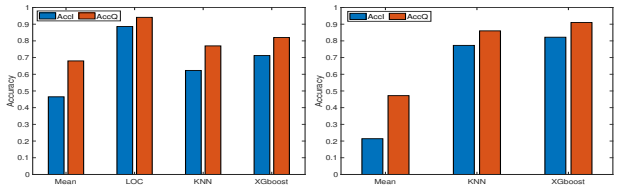
(Seconds)	Min	Max	Avg	Avg Speed Up
WiFi	3.9	46.3	19.2	1200X
WiFi-Large	6.2	74.5	33.6	19607X

and estimated 183 hours in WiFi-large [12] data set, respectively. In contrast, in Table 5, ZIP has only 19.2 and 33.6 seconds of run time in WiFi and WiFi-large data sets, and it speeds up the offline approach by 1200X and 19607X in WiFi and WiFi-large data sets.

Experiment 2: ZIP VS QTC-Eager. In Table 4 we report the runtime (in seconds) and number of imputations, i.e., the number of missing values imputed for *QTC-Eager*, *ZIP-lazy* and *ZIP-adaptive* approaches using the *random* query set. Note that LOC is applied on WiFi and Smart-Campus since it is not applicable on CDC data set. In Fig 10, we compare the performance of QTC-Eager with ZIP-lazy and ZIP-adaptive. We show the percentage of run time and number of imputations performed by ZIP-lazy and ZIP-adaptive compared to QTC-Eager. *time ratio* is the run time of ZIP-lazy (or ZIP-adaptive) divided by the time of QTC-Eager times 100 (percentage). Similarly, # *Imp ratio* is the percentage of imputation numbers.

We make several observations. First, ZIP-lazy and ZIP-adaptive perform similarly and they both outperform QTC-Eager by around 20x when expensive imputations are used, which demonstrate that delaying imputations significantly improves performance when imputations are expensive. Second, when cheap imputations are used such as Mean imputation, ZIP-adaptive tends to impute data first since doing so will potentially save the query processing time by reducing temporary tuples, and thus has similar imputations as QTC-Eager. This shows that the decision function in ZIP-adaptive works correctly and is able to actively adjust its decision based on the cost of imputations. Third, ZIP-lazy requires slightly less imputations than ZIP-adaptive since it always delays imputation to the end of processing, while ZIP-adaptive performs similarly as ZIP-lazy in WiFi and CDC and ZIP-adaptive slightly outperforms ZIP-lazy in Smart Campus due to the more complex join workload (higher join selectivity). Fourth, as expected, when learning approaches are used whose training time dominates the inference costs (e.g., as in XGboost), reducing number of imputation will not offer a big improvements. For instance, ZIP-lazy imputes 11.6% of imputations needed for QTC-Eager, but it takes 89% run time of QTC-Eager (i.e., saving only 11%).

Experiment 3: Quality of Query Answer. Quality of query answer depend upon the imputation method used and the query itself. Figure 11 plots the accuracy of imputation methods (Acc_I) and that



(a) WiFi (b) CDC

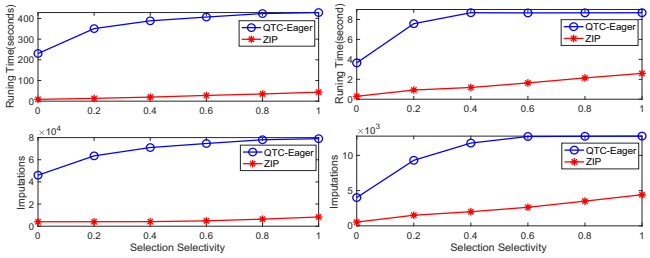
Figure 11: Accuracy of Imputations and Query Answer

of the corresponding query answers (Acc_Q). Acc_I is the percentage of correctly imputed values. For aggregate queries, Acc_Q is measured as $|\frac{\mathcal{A}^{\mathcal{T}}-\mathcal{A}}{\mathcal{A}^{\mathcal{T}}}|$, where $\mathcal{A}^{\mathcal{T}}$ is the true answer and $\mathcal{A}$ is the answer returned using an imputation method. For set based queries Acc_Q is measured using Jaccard similarity. Using different imputations with different accuracy, the accuracy of query answers is also different. The experiment above highlights the impact of choice of imputation method on query answer quality. To measure the *difference* between answers returned by ZIP and the offline approach, we use Symmetric-Mean-Absolute-Percentage-Error (SMAPE) [39] (also used by ImputeDB [17]). SMAPE is computed as a tuple-wise absolute percentage deviation between ZIP & offline approach. Since ZIP-lazy and ZIP-adaptive all return exactly the same answers as offline, the SMAPE value for all is 0.

Experiment 4: Query Selectivity Effects. We use the query template below to generate *low-selectivity* and *high-selectivity* query workloads: SELECT a, AVG(b) FROM $R_1, ..., R_n$ WHERE $[Pred_J]$ $[Pred_S]$ GROUP BY a., where $Pred_J$ and $Pred_S$ are join and selection predicates. We varied the selectivity of each selection predicate as 0, 0.2, 0.4, 0.6, 0.8, 1, and the selectivity of join predicate is set to be low and high by modifying the matching numbers of joined attribute values. KNN is applied in CDC data set, while in WiFi and Smart-Campus data set, LOC is used to impute location and occupancy, and Mean-value is used to impute other missing values. In CDC and WiFi data set, we report the effect from selectivity of selection predicates in Fig 12, and the effects from both join and selection selectivity are evaluated in synthetic data set in Fig 13. In this and later experiments, if no ambiguity, we call ZIP as the hybrid of ZIP-lazy and ZIP-adaptive – ZIP always uses ZIP-adaptive for non-blocking imputations and ZIP-lazy for blocking imputations.

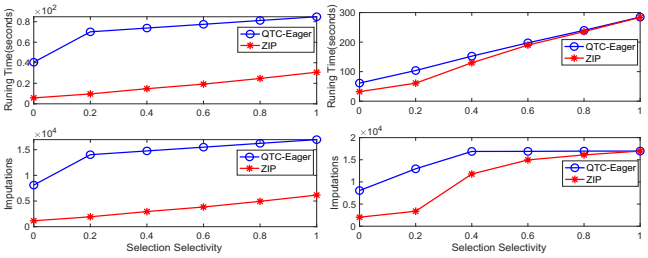
Number of imputations and running time increases for both QTC-Eager and ZIP when selectivity of predicates increases, though ZIP has considerably lower overhead and running time at all selectivity levels. In CDC, since KNN is costly and join operations are selective, ZIP delays imputations in selection operators. Join predicates help eliminate tuples reducing imputations needed thus reducing cost.

[12]We stop cleaning at 10 hours and the offline approach only imputes around 5% missing data.



(a) WiFi Data Set. (b) CDC Data Set.

Figure 12: Selectivity Effects on Real Data Set.



(a) Low Join Selectivity. (b) High Join Selectivity.

Figure 13: Selectivity Effects on Synthetic Data Set.

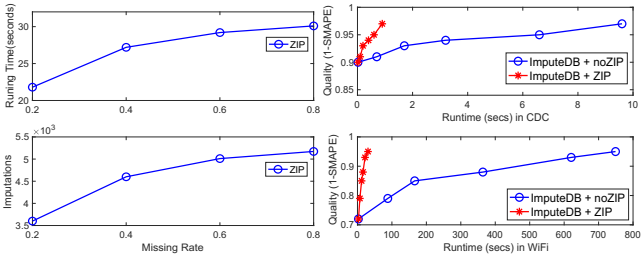


Figure 14: Missing Rate Effects.

Figure 15: ZIP + ImputeDB.

In WiFi where join attributes (e.g., location) have missing values, instead of paying expensive imputations (e.g., LOC) to impute all missing values in join attributes as QTC-Eager, ZIP delays the imputations of the missing values in join and allow the downstream operators to wisely choose cheap imputations (e.g., Mean) for missing values that are not in join attributes to help eliminate tuples if the corresponding predicates are selective. In synthetic data set, when the join operators are highly selective, (i.e., selectivity is low) as in Fig 13(a), ZIP delays imputing missing values in selection operators to exploit join operators to help eliminate most tuples and thus save most imputations. When join is less selective (i.e., more result tuples in Fig 13(b)) but the selection predicates are selective, ZIP might choose to partially delay the imputations in join attributes and the downstream operators (could be selective due to other selections) to help remove tuples to avoid expensive imputations in join attributes (e.g., location). However, when the selection predicates are also less selective and query processing overhead is higher than imputation costs, ZIP prefers imputing missing values immediately same as QTC-Eager.

Experiment 5: The effect of Missing Rates. Figure 14 shows how missing rates affect the query performance. With the increasing missing rates, the runtime and imputation times taken by ZIP slightly increase and tend to converge, since the amount of imputations needed to answer a given query depends on the selectivity of the query. For the set of queries with fixed selectivities, their performance will not be dramatically affected by the number of missing data in the data, which demonstrates the robustness of ZIP.

Experiment 6: ZIP with ImputeDB plans. We investigate ZIP using ImputeDB-generated query plans. ImputeDB [17] adds *impute* and *drop* operators to a query plan based on a parameter α ($0 \leq \alpha \leq 1$) that balances efficiency and quality (see Section 2). Higher the value of α, more drop operators are used causing more tuples with missing value to be dropped. To incorporate ImputeDB plans in ZIP, we added the drop operators that mirror the drop operator in ImputeDB. Given a ImputeDB plan, *impute* operators are treated as regular ZIP impute operator that could be evaluated lazily based on decision function but drop operator checks if a tuple contains missing values in the appropriate attribute and, if so, drops the tuple. We generated several plans based on varying α from 0 to 1 and execute the plan both in ImputeDB and in ZIP modified as above to support ImputeDB plans with *drop* operator.

Fig 15 shows the average quality (1-SMAPE) versus runtime of queries for the CDC and WiFi data using the KNN imputation approach on *random* query sets for ImputeDB plans with and without ZIP optimization. Each line plot shows 6 points corresponding to $\alpha = 1, 0.8, 0.6, 0.4, 0.2$, and 0 from left to right. The improvements are 10x to 25x when $\alpha = 0$ (i.e, when ImputeDB plan optimizes for quality, choosing *impute* over *drop*). When α increases, the relative improvements due to ZIP optimization reduces. This is expected since ZIP optimization only applies to tuples that are imputed. With increasing α more tuples are dropped reducing the need for ZIP optimization. When the cost-quality tradeoff across ImputeDB plans with and without ZIP optimization, to achieve the same level of quality, plans with ZIP optimization require significantly less time. Furthermore, ImputeDB plans when executed with ZIP optimization achieve significantly higher quality in the same amount of time spent in query processing. Thus, execution of ImputeDB plans with ZIP significantly dominates the ImputeDB plans without ZIP. The experiment clearly establishes that even in use cases where we explore cost-quality tradeoffs, ZIP-optimization can help improve systems such as ImputeDB that explore such tradeoffs, and ZIP and ImputeDB are complementary approaches integration of which provides a powerful query time imputation approach.

9 CONCLUSION

This paper studies query-driven missing value imputation and proposes ZIP, a technique to intermix query processing and missing value imputation to minimize query overhead. Specifically, ZIP co-optimizes imputation and query processing cost, and proposes a new implementation based on outer join to preserve missing values. Real experiments shows that ZIP combined with the state-of-the-art technique has 10x to 25x improvement, and ZIP outperforms the offline approach by up to 19607x in the real data set.

ACKNOWLEDGMENTS

This material is based on research sponsored by HPI under Agreement No. FA8750-16-2-0021, and it is partially supported by NSF Grants No. 1527536, 1545071, 2032525, 1952247, 1528995 and 2008993.

REFERENCES

[1] 2013-2014. *Center for Disease Control. National Health and Nutrition Examination Survey. https://wwwn.cdc.gov/nchs/nhanes/ContinuousNhanes/Default.aspx?BeginYear=2013.*

[2] 2017. *Github Codebase of ImputeDB. https://github.com/mitdbg/imputedb.git.*

[3] 2019. *6.830 Lab 1: SimpleDB. http://db.csail.mit.edu/6.830/assignments/lab1.html.*

[4] 2021. *Histogram-Based Gradient Boosting Ensembles. https://machinelearningmastery.com/histogram-based-gradient-boosting-ensembles/.*

[5] 2021. *https://en.wikipedia.org/wiki/Bloom_filter.*

[6] 2022. *AdventureWork Data Set. https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks.*

[7] 2022. *Cloud4WI. https://cloud4wi.com/proximity-messaging/.*

[8] 2022. *https://scikit-learn.org/stable/modules/generated/sklearn.impute.KNNImputer.html.*

[9] 2022. *IndoorAtlas. https://www.indooratlas.com/solutions/indooratlas-location/.*

[10] 2022. *WIACOM. https://wiacom.ai/.*

[11] 2023. *Technical report, ZIP: Lazy Imputation during Query Processing.*

[12] Hotham Altwaijry, Dmitri V Kalashnikov, and Sharad Mehrotra. 2013. Query-driven approach to entity resolution. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1846–1857.

[13] Hotham Altwaijry, Sharad Mehrotra, and Dmitri V Kalashnikov. 2015. Query: A framework for integrating entity resolution with query processing. *Proceedings of the VLDB Endowment* 9, 3 (2015), 120–131.

[14] Manos Athanassoulis, Kenneth S Bøgh, and Stratos Idreos. 2019. Optimal column layout for hybrid workloads. *Proceedings of the VLDB Endowment* 12, 13 (2019), 2393–2407.

[15] Parikshit Bansal, Prathamesh Deshpande, and Sunita Sarawagi. 2021. Missing value imputation on multidimensional time series. *arXiv preprint arXiv:2103.01600* (2021).

[16] Lane F Burgette and Jerome P Reiter. 2010. Multiple imputation for missing data via sequential regression trees. *American journal of epidemiology* 172, 9 (2010), 1070–1076.

[17] José Cambronero, John K Feser, Micah J Smith, and Samuel Madden. 2017. Query optimization for dynamic imputation. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1310–1321.

[18] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[19] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Discovering denial constraints. *Proceedings of the VLDB Endowment* 6, 13 (2013), 1498–1509.

[20] Xu Chu, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, Nan Tang, and Yin Ye. 2015. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1247–1261.

[21] Nhan Cach Dang, María N Moreno-García, and Fernando De la Prieta. 2020. Sentiment analysis based on deep learning: A comparative study. *Electronics* 9, 3 (2020), 483.

[22] Wenfei Fan, Jianzhong Li, Shuai Ma, Nan Tang, and Wenyuan Yu. 2012. Towards certain fixes with editing rules and master data. *The VLDB journal* 21, 2 (2012), 213–238.

[23] Dhrubajyoti Ghosh, Peeyush Gupta, Sharad Mehrotra, Roberto Yus, and Yasser Altowim. 2022. JENNER: just-in-time enrichment in query processing. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2666–2678.

[24] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 805–815.

[25] John W Graham. 2009. Missing data analysis: Making it work in the real world. *Annual review of psychology* 60 (2009), 549–576.

[26] Joseph M Hellerstein. 2008. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)* 25 (2008).

[27] Melanie Herschel, Mauricio A Hernández, and Wang-Chiew Tan. 2009. Artemis: A system for analyzing missing answers. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1550–1553.

[28] Hosagrahar Visvesvaraya Jagadish, Nick Koudas, S Muthukrishnan, Viswanath Poosala, Kenneth C Sevcik, and Torsten Suel. 1998. Optimal histograms with quality guarantees. In *VLDB*, Vol. 98. 24–27.

[29] Mourad Khayati, Ines Arous, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. ORBITS: online recovery of missing values in multiple time series streams. *Proceedings of the VLDB Endowment* 14, 3 (2020), 294–306.

[30] Mourad Khayati, Alberto Lerner, Zakhar Tymchenko, and Philippe Cudré-Mauroux. 2020. Mind the gap: an experimental evaluation of imputation of missing values techniques in time series. In *Proceedings of the VLDB Endowment*, Vol. 13. 768–782.

[31] Zuhair Khayyat, Ihab F Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. Bigdansing: A system for big data cleansing. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1215–1230.

[32] Won Kim, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. 2003. A taxonomy of dirty data. *Data mining and knowledge discovery* 7, 1 (2003), 81–99.

[33] Manikanta Kotaru et al. 2015. Spotfi: Decimeter level localization using wifi. In *SIGCOMM*. ACM.

[34] Peng Li, Xi Rao, Jennifer Blase, Yue Zhang, Xu Chu, and Ce Zhang. 2021. CleanML: a study for evaluating the impact of data cleaning on ml classification tasks. In *36th IEEE International Conference on Data Engineering (ICDE 2020)(virtual)*. ETH Zurich, Institute for Computing Platforms.

[35] Zan Li et al. 2015. A passive wifi source localization system based on fine-grained power-based trilateration. In *WoWMoM*. IEEE, 1–9.

[36] Wei-Chao Lin and Chih-Fong Tsai. 2020. Missing value imputation: a review and analysis of the literature (2006–2017). *Artificial Intelligence Review* 53, 2 (2020), 1487–1509.

[37] Yiming Lin et al. 2021. Locater: cleaning wifi connectivity datasets for semantic localization. *Proceedings of the VLDB Endowment* 3 (2021), 329 – 341.

[38] Tongyu Liu, Ju Fan, Yinqing Luo, Nan Tang, Guoliang Li, and Xiaoyong Du. 2021. Adaptive data augmentation for supervised learning over missing data. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1202–1214.

[39] Spyros Makridakis and Michele Hibon. 2000. The M3-Competition: results, conclusions and implications. *International journal of forecasting* 16, 4 (2000), 451–476.

[40] Xiaoye Miao, Yangyang Wu, Lu Chen, Yunjun Gao, Jun Wang, and Jianwei Yin. 2021. Efficient and effective data imputation with influence functions. *Proceedings of the VLDB Endowment* 15, 3 (2021), 624–632.

[41] Zhixin Qi, Hongzhi Wang, Jianzhong Li, and Hong Gao. 2018. FROG: Inference from knowledge base for missing value imputation. *Knowledge-Based Systems* 145 (2018), 77–90.

[42] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. Holoclean: Holistic data repairs with probabilistic inference. *arXiv preprint arXiv:1702.00820* (2017).

[43] Shaoxu Song, Aoqian Zhang, Lei Chen, and Jianmin Wang. 2015. Enriching data imputation with extensive similarity neighbors. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1286–1297.

[44] Nitin Thaper, Sudipto Guha, Piotr Indyk, and Nick Koudas. 2002. Dynamic multidimensional histograms. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 428–439.

[45] Chen Ye, Hongzhi Wang, Jianzhong Li, Hong Gao, and Siyao Cheng. 2016. Crowdsourcing-enhanced missing values imputation based on Bayesian network. In *International Conference on Database Systems for Advanced Applications*. Springer, 67–81.

[46] Chen Ye, Hongzhi Wang, Wenbo Lu, and Jianzhong Li. 2020. Effective Bayesian-network-based missing value imputation enhanced by crowdsourcing. *Knowledge-Based Systems* 190 (2020), 105199.

[47] William Young, Gary Weckman, and W Holland. 2011. A survey of methodologies for the treatment of missing values within datasets: Limitations and benefits. *Theoretical Issues in Ergonomics Science* 12, 1 (2011), 15–43.