Hardware-Efficient Data Imputation through DBMS Extensibility

Hubert Mohr-Daurat
Imperial College London
h.mohr-daurat19@imperial.ac.uk

Georgios Theodorakis*
Neo4j
george.theodorakis@neo4j.com

Holger Pirk
Imperial College London
hlgr@ic.ac.uk

ABSTRACT

The separation of data and code/queries has served Data Management Systems (DBMSs) well for decades. However, while the resulting soundness and rigidity are the basis for many performance-oriented optimizations, it lacks the flexibility to efficiently support modern data science applications: data cleansing, data ingestion/augmentation or generative models. To support such applications without sacrificing performance, we propose a new logical data model called *Homoiconic Collection Processing (HCP)*. HCP is based on a well-known Meta-Programming concept called *Homoiconicity* (a unified representation for code and data).

In a DBMS, HCP supports the storage of "classic" relational data but also allows the storage and evaluation of code fragments we refer to as "Homoiconic Expressions". Homoiconic Expressions enable applications such as data imputation *directly in the database kernel*. Implemented naïvely, such flexibility would come at a prohibitive cost in terms of performance. To make HCP performance-competitive with highly-tuned in-memory DBMSs, we develop a novel storage and processing model called *Shape-Wise Micro-batching (SWM)* and implement it in a system called BOSS. BOSS is performance-competitive with high-performance DBMSs while offering unprecedented extensibility. To demonstrate the extensibility, we implement an extension for impute-and-query workloads: BOSS outperforms state-of-the-art homoiconic runtimes and data imputation systems by two to five orders of magnitude.

PVLDB Reference Format:
Hubert Mohr-Daurat, Georgios Theodorakis, and Holger Pirk. BOSS - An Extensible DBMS Architecture. PVLDB, 17(11): 3497 - 3510, 2024.
doi:10.14778/3681954.3682016

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

1 INTRODUCTION

More than five decades of research and development have honed Database Management Systems (DBMSs) into highly efficient, easy-to-use pieces of software with a well-defined logical model. The soundness of the relational model served these systems well when the primary challenges were correctness and performance. Modern data processing requirements, however, have outgrown such rigid models: data science pipelines are increasingly complex, involving data ingestion from heterogeneous sources, cleaning, integration, transformation, loading, querying, use for model training/inference, and visualization for end users. In typical data processing pipelines, a sequence of dedicated and specialized systems implement each of these processing steps. This approach leads to two fundamental problems. First, data must be migrated from system to system along the pipeline, which incurs significant overhead for conversion, transfer and integration. Second, it requires the costly implementation, deployment, and operation of specialized systems.

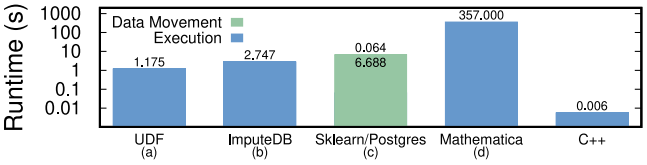

Figure 1: Executing TPC-H Q6 with 10% imputed values

To illustrate the challenges of modern data processing, consider a common first step in a data science pipeline: value *imputation*, i.e., the derivation of missing values from present ones. Most DBMSs offer no native support for such "data cleansing" operations. While SQL prescribes the concept of `NULL` to represent missing values, it defines no coherent way to interpret them beyond coarse rules like "NULL values must not contribute to aggregates" [25]. Users, therefore, face the choice of four approaches to data imputation: (a) extend the DBMS's functionality using User-defined Functions (UDFs); (b) extend the kernel; (c) use a dedicated cleansing system [18, 45]; or (d) process data in a data-science-oriented programming language such as Julia [6] or Wolfram Mathematica [54].

Each of these approaches faces unique performance challenges. To illustrate these, we implemented a combined imputation & querying pipeline for typical representatives of each approach. We ran Query 6 of the TPC-H dataset with a scale factor of 0.1 and 10% randomly "NULLed-out" `L_DISCOUNT` values and imputed them during query processing as the mean of the column. The results in Figure 1 demonstrate that each approach is at least two orders of magnitude slower than the performance target in plain C++: (a) a UDF, even using a commercial database that is known for good UDF support, prevent effective optimization and execution; (b) ImputeDB [9] demonstrates the overhead of a DBMS architecture that supports kernel-level extensions; (c) data imputation using SckiKit-learn [37, 48] connected to PostgreSQL [40] spends virtually all of the time loading dirty data from Postgres and copying cleaned data back; (d) Wolfram Mathematica[1] suffers from the well-known interpretation overhead of dynamic language runtimes [27, 52].

However, the problem is harder than just replacing `NULL`s with generated values. High-quality data cleaning requires capturing not just *that* a value is missing but *why* it is missing. The conventional

*Work done while at Imperial College London

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3682016

[1]Wolfram advertises data imputation as a key feature of Mathematica [53]

```
(Table (Schema 'KEY  'SHIPDATE 'DISCOUNT  'TAX)
  (Tuple         1   "96-03-13" (Mean)    .10)
  (Tuple         2   "96-04-12"  .04      .08)
  (Tuple         3   "96-01-29" (Mean)    .06)
  (Tuple      (GenID) 'OnHold    .09  (If (> 'SHIPDATE
                                      "96-06-01") .04 .06))
  (Tuple      (GenID) 'OnHold    .10  (If (> 'SHIPDATE
                                      "96-06-01") .02 .03)))
```

Figure 2: A Relation as Homoiconic Expression

way to capture the why is to represent missing values as NULLs with a "tag" value (e.g., an enum) in a separate column (we will call this "tagged nulls"). Imputation logic is encoded into queries through rewriting. This approach suffers from several problems: first, it requires extra columns to hold the tags for every column that *might* have missing values (i.e., virtually every column). Second, tag interpretation through query rewriting is non-trivial, costly and incurs runtime overhead (even if no values are missing). Third, to ensure that imputed results are consistent even if the imputation process changes, versions of the imputation logic must be encoded into the tags, further complicating rewriting. Fourth, tagged nulls cannot encode payloads such as numeric parameters for the imputation process. Finally, they do not support composition to encode complex processes (such as conditional branches). These problems lead to significant limitations in real use cases: exception handling for edge device data logging cannot capture complex reasons for failure with mere tagged-nulls; Model-generated data can only be stored as values and not as a generation "process", preventing, e.g., consistent re-generation when the model evolves; Personal data deletion for legal compliance (e.g., GDPR or CCPA) cannot keep any record that data ever existed or why it was deleted without a mechanism to embed a "process" into the data model.

While we plan to study these applications in the future, in this paper, we focus on laying the groundwork for the representation of processes in a database. We argue that the rigidity of the relational data & processing model prevents the effective capture of processes, while the flexibility of dynamic languages comes with execution overhead unsuitable for high-performance data processing.

We define a "process" as any series of operations that generate, retrieve or transform data. Arguably, programmable computers and, by extension, DBMSs have a way of representing processes: binary code. However, code is effectively a black box: it can be evaluated but not (reasonably) manipulated. It is, thus, the diametrical opposite of (managed) data, which can be manipulated but not evaluated. We propose to embed processes as first-class citizens into Database Management Systems (DBMSs), allowing it to be manipulated like data (queried and modified) and executed on-demand.

Some programming languages (notably LISP [33] family languages like Clojure, Racket and Julia) support process as a first-class citizen through a paradigm called *homoiconicity*: data and code represented in a unified form that can be manipulated.

To illustrate homoiconicity in the context of our work, consider Figure 2: an expression in LISP syntax representing a table with schema (i.e., metadata) and tuples (i.e., data). The tuple attributes, however, are either data values (integers, floats, strings) or expressions (i.e., code, indicated by the parentheses). While we discuss the details in the next section, the example illustrates the embedding of process in the data in a form that can be manipulated as well as executed: (GenID), e.g., generates keys on access and (Mean) indicates that the values shall be imputed as the mean of the column's known values.

We will demonstrate the utility of homoiconicity in DBMSs, using data imputation as a motivating case (leaving generative models, provenance, data integration and others for future work). Focusing on efficiency, our objective is what we call Pay-As-You-Go (PAYG) homoiconicity: processing non-homoiconic data as fast as modern in-memory DBMSs while suffering only minimal overhead and only proportional to the number of homoiconic expressions in the database. To this end, we make the following contributions:

- We introduce "Homoiconic Collection Processing", a novel *logical* data and processing model breaking the strict separation of data and code. The key idea is a collection-oriented data model that allows expressions as attribute values. We show that it strikes an appropriate balance between flexibility and performance.
- To efficiently process homoiconic collections, we develop a novel storage & processing model called *Shape-wise Microbatching (SWM)*. SWM enables memory-efficient storage and CPU-efficient (micro-batched/vectorized) processing.
- To show the practicality of the approach, we develop a homoiconic DBMS called BOSS. Its performance rivals modern in-memory DBMSs while also processing homoiconic expressions. It significantly outperforms state-of-the-art homoiconic runtimes.
- To demonstrate the advantages of homoiconicity for data processing, we developed a data imputation module for BOSS. It allows user-defined data imputation at modern DBMS performance – orders of magnitude faster than competing systems.

2 BACKGROUND

In this section, we introduce the data management and programming language concepts we expand on in the rest of this paper.

2.1 Homoiconic Programming

While classic programming languages distinguish compile-time (code) and runtime (data) aspects of a program, *Homoiconicity* blurs these boundaries; Homoiconic languages allow pieces of code to be manipulated at runtime, i.e., treated as data, but also to be (compiled and) executed at runtime, i.e., treated as code.

In such languages, the fundamental building blocks of expressions are *atoms*. There are either *simple atoms*, i.e., the basic datatypes found in virtually any programming language (such as the ints, floats and strings shown in Figure 2) or *symbols* (marked with a single quote prefix, such as 'OnHold in Figure 2). Symbols are identified by their name (a string) and generalize the concept of variables in imperative programs. However, while variables in imperative programs must be backed up by memory and hold a value at runtime, symbols may be undefined.

On top of symbols, homoiconic languages allow the construction of complex expressions (a.k.a. *symbolic expressions* or *s-expressions*). (Plus 1 2 3), e.g., represents 1 + 2 + 3 while (list 1 2 3) represents the list containing 1, 2 and 3. While conceptually, the first represents code, the second data, the difference is merely their

first element (a.k.a. the *head*). S-expressions can be evaluated as code or "held" in their unevaluated form to be manipulated as data.

Interestingly, most users are already familiar with homoiconic data processing without noticing: virtually all spreadsheet systems [2, 3, 24, 34] allow storing formulae in cells that are eagerly evaluated when entered. While the scalability issues of traditional spreadsheets are well known [5], logically, homoiconic collections are a scalable extension of relations with spreadsheet-like concepts.

Homoiconic languages such as Wolfram Mathematica, Clojure or Racket can store and evaluate s-expressions representing a database like the one in Figure 2. However, their implementation does not scale: they cannot efficiently process the tuples due to the interpretation overhead to evaluate each expression one by one.

2.2 Decomposed Storage and Bulk Processing

To store and evaluate homoiconic data without the overhead of typical homoiconic execution, we build on state-of-the-art in-memory data storage and analytical query processing: the storage model of choice is the *Decomposed Storage Model* (DSM) [15], popularized by MonetDB [7]. Storing tuples' attributes in columns provides high bandwidth efficiency through data locality. Combined with (function-call-free) bulk operators, the model also yields high CPU efficiency. Building on bulk processing, (X100-style) micro-batching avoids the cost of materializing intermediate results to main memory [57] by processing columns as cache-resident micro-partitions.

2.3 Data Imputation in Databases

As this work focuses on data imputation, let us examine existing data imputation solutions in the context of DBMSs. The most common approach is to use external data cleansing systems [12, 14, 45, 46]. The user must specify when data cleansing is performed and manually transfer data between systems, which causes data movement overhead. The alternative solutions are to implement data imputation as a database kernel feature [9] including deferred imputation techniques to optimize query performance [32], or as User-Define Functions (UDFs), taking advantage of advanced optimization strategies [21, 43, 47]. Similar to UDFs, standard SQL stored procedures can be used for imputation but only on the base table rather than on the fly during query evaluation. All these in-DBMS solutions suffer from the execution overhead demonstrated in Section 1 and do not have the flexibility of dynamic language runtimes: to mark missing values, these imputation strategies store labels: either as "reserved" values in the missing attributes' column or separate columns. The "reserved-values" approach requires a mechanism to ensure that the reserved values are excluded from the domain of valid values and, even worse, does not support the representation parameters required for many imputation methods. Examples of such parameters are prompts for generative models, parameters for source-specific statistical distributions (e.g., device-specific mean and variance for normal distributions when acquiring sensor data) or references to other columns. Such parameters can be stored in separate columns when that strategy is adopted. However, that requires major schema changes: additional columns, not only per missing attribute but per imputation strategy applicable to that attribute. Ideally, data imputation would cover these cases without changes to the schema and without requiring application developers to change their code.

2.4 Running Example

To provide an intuition of how query processing, homoiconicity and data imputation interact, let us revisit the example database in Figure 2, which will serve as a running example throughout the paper: inspired by the TPC-H `LINEITEM` table, it stores orders of products with their shipping date and price adjustments (discounts and tax). However, some values in the `DISCOUNT` column are missing. The user has marked these for imputation by entering a `(Mean)` expression (the mean of the known values). In addition, the last two tuples' `SHIPDATE` have not been finalized yet, and to reflect it, the database stores these values as missing: they are substituted by the symbol `'OnHold`. In addition, because of new legislation that year, the tax rate changes based on the shipping date. To indicate this, the database stores the `TAX` values as expressions dependent on the missing `SHIPDATE` values: that is expressed as `(If (> 'SHIPDATE "96-06-01") $x $y)`. They are imputed at query time when the `SHIPDATE` value is known. If the symbol is not known at query time, the expression stays unevaluated as described in Section 3.5.

As shown in these examples, imputation expressions can be composed of operators and arguments which define the precise tree of operations to be executed by the kernel to calculate the missing values. When it is desired to change the imputation logic after tuples have been inserted, a declarative form is used, such as the expression `(Mean)`, which executes a custom operator or a function specified only at query time. The exact semantics for the expressions will be clarified later in the paper.

3 HOMOICONIC COLLECTIONS

As illustrated in Section 1, the complexity of modern data science pipelines calls for the flexibility of homoiconic runtimes. However, state-of-the-art homoiconic programming approaches have such an interpretation overhead that they scale too poorly for big data processing. To strike a balance between the flexibility of homoiconic programming and the performance of bulk in-memory processing, we propose a novel paradigm called *Homoiconic Collection Processing (HCP)* and define three objectives that drive its design:

- **Expressivity** enables a data scientist to store all aspects of their workflow (data and code) in a representation that supports the needs of their complex pipelines. The example in Figure 2 illustrates the need to store data representing unknown values as well as data representing any processes to be executed when a tuple is processed. Unknown values include values that will be known later (e.g., `'OnHold` evaluating to a shipping date only when, in the future, the user substitutes 'OnHold with the known date) and values that, by nature, are known only at query time (e.g., `(> 'SHIPDATE "96-06-01")` which operates on the tuple attribute `SHIPDATE`). Processes include lazy-loaded or lazy-generated data (e.g., `(GenID)`) and the imputation of missing data (e.g., `(Mean)`).
- **Extensibility** allows users to define the "semantics" of expressions by implementing new operators. In Figure 2, the expression `(Mean)` is a declarative form representing a specific imputation method to apply to missing data: the user must be able to implement such an operator as a kernel extension easily.

```
(Group (Select 'LINEITEM (Where (> 'TAX .07))) 'Count)
```

Figure 3: A Relational Query as Homoiconic Expression

$$
\begin{aligned}
\text{HRExpr } \mathcal{E} &::= \mathcal{A} \mid \mathcal{S} \mid C \\
\text{Atoms } \mathcal{A} &::= Bool \mid Int \mid Float \mid String \\
\text{Symbols } \mathcal{S} &::= \langle str \rangle, \quad str \in String \\
\text{ComplexHRExpr } C &::= \langle head, arg_1, ..., arg_n \rangle, \qquad head \in \mathcal{S}; \forall i, arg_i \in \mathcal{E}
\end{aligned}
$$

Figure 4: Formal Representation of an H-R Expression

- **Performance** allows users to scale data science pipelines to large datasets. It also enables interactive data exploration, which is crucial to modern data science. Consequently, performance must not be sacrificed for homoiconicity: a homoiconic DBMS must be carefully engineered to ensure efficient runtime in the presence of homoiconic expressions. As overhead for expression evaluation cannot be avoided entirely, we define our performance objective as *Pay-As-You-Go* (PAYG): the system shall process conventional data as fast as modern in-memory DBMSs and suffer (minimal) overhead only proportional to the number of stored expressions.

Let us, in the following, break down these objectives into technical challenges and propose ideas to address each of them.

3.1 Head-Restricted Expressions

The first challenge is the **complexity of expression representation**. The *Expressivity* objective can naïvely be achieved by representing complex expressions as trees with atoms as leaves. However, representing a fully generic expression tree like that is memory-inefficient and incurs significant interpretation overhead. To satisfy the *Performance* objective, the expression must be represented as a concise in-memory data structure that can be efficiently processed.

To that end, we propose an expression model slightly more restricted than classic "LISP-style" expressions. Specifically, LISP expressions allow (nested) expressions not only as arguments but also as heads of expressions. This is used, e.g., to represent functional constructs like lambdas. For the kind of systems we aim to implement, this use is rare: the primary use of the expressions in data science systems is to execute operators from a pre-defined set of kernel operators, e.g., evaluate imputed expressions like (`Mean`) or relational operators, such as `Select` and `Group` (Figure 3 shows a typical select and aggregate query encoded in s-expression syntax).

To exploit this focus on data science, we restrict the expressivity of complex expressions in a concept called *Head-Restricted Expressions* (H-R expressions). The formal definition of H-R expression is given in Figure 4: like classic expressions, H-R expressions are either atoms, symbols or complex H-R expressions. While the arguments of the complex H-R expressions can be any H-R expressions, the critical restriction pertains to the head: *Complex H-R expressions allow only Symbols as the head.* This restriction enables the efficient storage and evaluation of expressions without impacting expressivity: while it prevents lambdas as expression heads, functional constructs are still possible by implementing an `Apply` operator, which dynamically replaces the head of an expression.

3.2 Homoiconic Collection Representation

The second performance challenge is the **overhead of evaluating a collection of homoiconic data**. Logically representing homoiconic relations as in Figure 2 (i.e., as expressions of the form (`Table (Tuple ...)`)) is convenient and extensible. Naïvely representing these as in-memory trees, however, is unsuited as physical representation as the unpredictable and fragmented structure adds substantial interpretation overhead (i.e., violates the *Performance* objective). However, virtually all data science applications process data in collections. As we focus on relational data, these collections are homoiconic relations, i.e., relations of "homoiconic tuples" (tuples of H-R expressions)[2]. To exploit this specialization, we define a data model that imposes a common structure onto a collection of H-R expressions. We call this data model *Homoiconic Collections*. The key idea behind homoiconic collections is to represent tuples as *arrays* of H-R expressions. In practice, it is sensible to expect that many tuples in a collection will have the same or similar structure. We demonstrate in Section 4 how this insight enables PAYG homoiconicity while providing the flexibility to handle the cases illustrated in Section 2.3. It even enables the specialization to specific data models: in relational data, e.g., the storage of the `Table`- and `Tuple`- heads is elided, and only the arguments of `Tuple` expressions are stored (in plain arrays).

3.3 Custom Operators and Data Types

The third challenge is the **complexity to extend the kernel with new operators** as required by the *Extensibility* objective: the design must be flexible enough to implement new operators used in expressions (e.g., imputation operators) and extend the type system with new data types needed for these operators (e.g., to integrate external data structures like tensors or pre-trained models).

To provide extensibility of the operator set, we define an operator using a generic interface: the *HCP-Operator*. It is a function that takes H-R expressions as arguments and returns an H-R expression. However, HCP-Operators can restrict the set of acceptable types for each argument to atom types, symbols or complex H-R expressions. For example, the implementation of the operator `Plus` defines, as the set of acceptable argument types, any pair of numerical atoms. At runtime, an expression is evaluated opportunistically: the expression (`Plus 1 2`), e.g., matches the signature and is evaluated while the expression (`Plus "a" 1`) remains unevaluated.

3.4 Dynamic Typing

The fourth challenge is the **uncertainty of the type resulting from evaluating an expression**. Only after an expression is evaluated is the result type determined. This leads to a challenge for the HCP model: not only can attribute types vary within a column but the types are only known at query evaluation time. Even in the simplest case where a column type is fixed at the kernel level, columns can hold values of two types: an atom or an expression that yields an atom of the fixed type. For example, `5` is not the same type as (`+ 3 2`) even though they yield the same value. The distinction is important for type-checking when storing homoiconic and atomic values within the same column, as in the example in Figure 2. The processing model must be designed with a mechanism

[2]Note that the principle is just as applicable to graphs or key-value pairs

to dispatch (efficiently) homoiconic and atomic values to operator implementation at runtime. As a consequence, the kernel does not require static-type information for the processing. However, we also assume that all imputation methods are well-behaved, which allows us to avoid performing static type checking. In practice, it would be possible to use an imputation method that misbehaves, i.e., not returning the expected type. This problem is outside the scope of this paper. However, it would be possible to implement an operator to perform a static-type check at the beginning of the query evaluation (we will keep this for future work). Finally, operators that take unevaluated expressions as arguments, e.g., an operator that orchestrates data imputation, must handle dynamic type dispatch within their implementation.

3.5 Partial Evaluation

The fifth challenge is the **processing of unevaluated expressions and symbols**. Symbols and expressions can remain unevaluated at runtime, e.g., when an operator is not implemented or when a symbol (such as `'OnHold`) has no defined value yet. However, this should not prevent the query evaluation altogether.

Instead, a system implementing the HCP model must support *partial-evaluation*: expressions shall be evaluated best-effort, recursively and depth-first. If one of the arguments of an expression cannot be evaluated (e.g., due to an undefined symbol), the other arguments are still evaluated (if possible). The evaluated expression is, in that case, constructed from the head of its unevaluated form and the (evaluated or unevaluated) arguments.

3.6 Homoiconic Operator Semantics

The partially evaluating nature of HCP operators complicates their semantics. Specifically, homoiconic expression processing deserves in-detail attention. Conceptually, we distinguish three cases:

The first case is when expressions exclusively contain symbols and subexpressions that have a definition (e.g., constants, symbols identifying relations or query parameters). In this case, the result of expressions is defined as in relational algebra: operators are stateless functions on the domain of "ordered multisets". While this allows the implementation of all relational operators, we currently restrict our work to Select, Project (including arithmetic), Group, Sort, TopN and Inner Join operators.

The second case is an operator that is "unassuming", i.e., it contains symbols that do not have a definition. In our implementation, all relational operators are unassuming. Such an operator evaluates its input into an equivalent query in a best-effort manner. If some subexpressions of the input are undefined, the operator yields a result that evaluates all fully defined subexpressions while leaving those with undefined subexpressions unevaluated. The result of the selection in Figure 3 on the table defined in Figure 2, e.g., would be a union of a table containing tuples 1 & 2 (which pass the predicate) and an unevaluated select on the table containing the tuples 4 & 5. Tuple 3 was eliminated as it fails the predicate.

The third case is an operator that is "assuming", i.e., it provides an "interpretation" of its input by substituting symbols or subexpressions for values based on "assumptions". An assuming operator interpreting the table in Figure 2 could, e.g., substitute `'OnHold` and `(GenID)` for specific values. In the context of this paper, all imputation operators are "assuming" but stateless. Upon evaluation, they implicitly receive the table they operate on and an identifier for the value they must produce (row and column) as input.

4 HCP STORAGE & PROCESSING MODELS

Established relational storage and processing models are insufficient to implement the HCP model efficiently. Expressions can be stored in most data processing systems only as binary objects. Manipulating arrays of binary objects leads to the same interpretation overhead that homoiconic languages face. To efficiently store and evaluate arrays of H-R expressions, we propose a novel paradigm we call *Shape-Wise Microbatching (SWM)*. SWM provides highly efficient PAYG homoiconic data management through careful storage & processing model co-design. We will cover them in this order.

4.1 Storage

The primary consideration when designing a storage model for HCP is to enable efficient processing. Prior work [38] has shown that N-ary storage is at odds with CPU-efficient bulk processing in relational DBMSs. This insight also applies to homoiconic expression data: the storage model for expressions must be carefully designed to support a CPU-efficient processing model.

In classic relational DBMSs, the Decomposed Storage Model (DSM) [15] has been shown to be appropriate to support bulk- or X100-style-processing engines [57]. However, no equivalent standard models store (potentially nested) expressions contiguously in memory in decomposed format. While document databases support trees, they are optimized to retrieve efficiently subtrees, not for expression evaluation. To efficiently support the storage and evaluation of homoiconic collections, we developed a novel storage scheme: *Shape-wise Partitioning & Decomposition (SWPD)*. Illustrated in Figure 5, SWPD comprises two conceptual steps we present here individually after defining the concept of a *shape*.

4.1.1 Shape of an Expression. We define the *shape of an expression* as the n-tuple of the expressions' head and (recursively) the shape of each expression argument if the expression is complex, i.e., not an atom. If it is an atom, the shape is simply the type of the atom. The shape of `(Plus 5 1.5)` is, e.g., $\langle Plus, Int, Float\rangle$ and the nested expression `(If (> 'SHIPDATE "96-06-01") .04 .06)` is $\langle If, \langle Greater, Symbol, String\rangle, Float, Float\rangle$.

4.1.2 Shape-wise Partitioning (illustrated in the left half of Fig. 5). Multiple expressions of the same shape can be stored in a contiguous memory region without per-expression structural information. This insight is the key to amortizing the overhead of evaluating expressions. However, as data can contain arbitrary expressions in the tuple attributes, the optimal construction of shape-homogeneous expressions requires (horizontally) partitioning collections of tuples such that all tuple attributes in a partition have the same shape. We call this process *Shape-Wise Partitioning*.

After applying SWP, the interpretation overhead is bounded upwards by the number of expressions in the database. It, therefore, satisfies our objective of PAYG homoiconicity. While the number of partitions grows, in the worst case, exponentially with the number of expression shapes, we follow established precedent [9, 56] by expecting the number of shapes/partitions to be significantly smaller than the number of expressions, i.e., 3 to 4 imputation strategies. Let

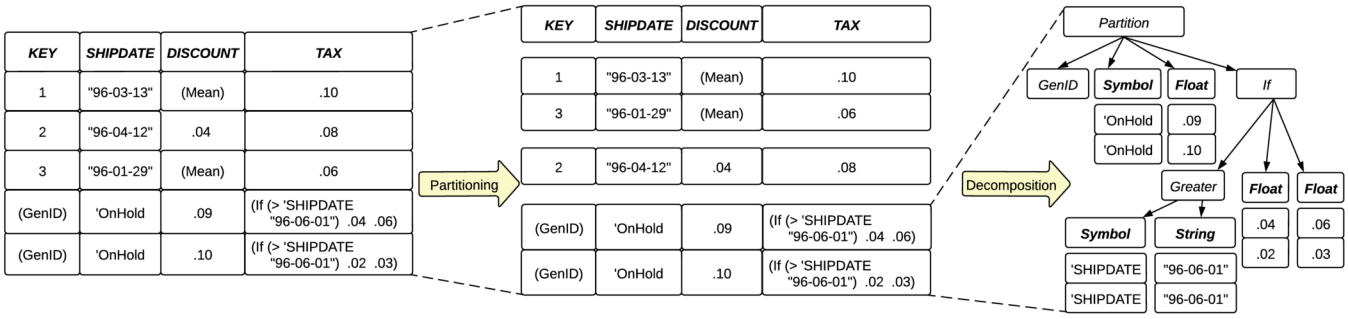

Figure 5: *Shape-wise Partitioning & Decomposition* Allows Efficient Storage of Homoiconic Expressions

us illustrate this rationale with an example: a table with ten columns, each of which contains four different expression shapes which indicate missing data, optionally encoding a mitigating strategy (such as `'NULL`, (`Mean`), (`HotDeck`) and (`RegressionTree`)[3]). The number of partitions in this example is bounded by 4^{10}, i.e., roughly 1 million. While this sounds high, most micro-batching query processors handle tens of millions of partitions in a single query without causing that substantial overhead. With similar calculation on the datasets found in previous work [9], the number of partitions is 134,664 on average, with a worst-case of 671,153 partitions.

4.1.3 Shape-wise Decomposition (illustrated in the right of Fig. 5). After partitioning data into shape-homogeneous partitions, the second step of SWD is their decomposition into columns: expressions are decomposed recursively up to their leaves (i.e., atoms). These atoms are stored in plain arrays of homogeneous data types. In Figure 5, two of the `DISCOUNT` values (.09 and .10) are stored in a ⟨*Float*⟩ column. The (`If (Greater ...)`) expressions in the `TAX` column are decomposed further to create ⟨*Symbol*⟩ and ⟨*String*⟩ columns holding the `Greater` operands and two ⟨*Float*⟩ columns holding the `If` operands. In order to avoid redundant information, expression heads are stored as per-partition metadata.

With this model, atoms are stored in a decomposed format for maximum CPU-efficiency. Let us now discuss the processing.

4.2 Processing

To minimize function call overhead while reducing materialization costs, we implement a processing model using cache-sized micro-partitions based on the MonetDB/X100 model [57]. We call this processing model *Shape-wise Microbatching (SWM)*. Unlike typical relational processing models, the expressions embedded in data require evaluation. In this section, we discuss the design and evaluation of operators, allowing consistent and efficient handling of queries and embedded expressions.

4.2.1 Operator Design. The Operator API we propose for SWM is primarily designed to handle data-intensive queries, such as the expression in Figure 3. The design must allow implementing relational operators (i.e., `SELECT`, `PROJECT`, `JOIN`, `GROUP BY`, `ORDER BY`), functional expressions used for aggregates, grouping/sorting functions, projectors and predicates, such as (`Where (> 'TAX .07)`) including the operators to calculate comparisons and arithmetic, and any other non-relational operator, such as the operator (`Mean`). To reduce

[3] strategies proposed in ImputeDB [9]

```
class Operator {
  // Implemented by each operator
  void consume(HRExpression arg0, ..., HRExpression argN);
  void close();
  // Common to all operators
  void pushUp(HRExpression output) final;
};
```

Figure 6: Operator API

the overhead of evaluating these functional expressions, we design operators to accept not just expressions but also arrays of expressions inspired by operator design in vectorized engines. This means that when evaluating an expression such as (`> 'TAX .07`), `'TAX` is evaluated to an array and the greater operator `>` is bulk-evaluated.

Traditionally, vectorized engines implement operators using a Volcano-style, i.e., pull-based interface. This interface is appropriate for statically typed data since every operator's implementation remains the same throughout the runtime of a query. In a dynamically typed engine like ours, however, the type of incoming data may change while an operator is executed, which requires a corresponding change of the operator. While Volcano-style operators can change their implementation dynamically, this requires a substantial amount of boilerplate code for dynamic type interpretation/dispatching, making the system harder to extend. To eliminate the need for dynamic-type-dispatching boilerplate code inside each operator, we propose to perform type dispatching outside the operators: the execution engine interprets dynamic types and invokes type-specific instances of operators. This requires a push-driven operator interface like the one illustrated in Figure 6.

Other than the push-driven interface, SWM operators follow the Volcano model: a type-specific `consume()` is called for each input batch and the constructed result is returned to the next operator.

4.2.2 Evaluation of Embedded Homoiconic Data. During query evaluation, the expressions embedded in data are not implicitly evaluated: a missing attribute value such as (`Mean`), e.g., would only be evaluated by placing a designated `Evaluate` operator in the query plan. While explicitly placing such operators may seem cumbersome, the flexibility to push `Evaluate` up or down the query plans provides optimization opportunities [9] as detailed in Section 5.2.4.

In many cases, e.g., projections or selections, these arrays of expressions are only passed along the relational operators without evaluation if, e.g., a predicate does not require the stored expressions' values. We will see how this design enables the effective processing of homoiconic data in Section 5.2.

When an array of expressions is evaluated, there is no assumption regarding the output type. The operator API defines a generic expression type for the output, and each operator's implementation may return any atomic or complex expression type. Consequently, when evaluating an array of expressions, the output is not necessarily a type-homogeneous array. SWM naturally handles this case by dynamically dispatching each input partition to the correct, statically typed operator (which may, in turn, produce a non-homogenous, shape-wise-partitioned output). Implementing operators as type-generic (C++) templates requires no additional implementation effort to handle different atomic types. It does, however, require operators to implement logic to support complex (i.e., partially evaluated) expressions. To handle the output of which parts are evaluated while parts are not, the approach used in other homoiconic languages is to output a union of two partitions: one containing evaluated and one unevaluated expression. We currently do not support this case but plan to do so in the future.

4.2.3 Symbol Substitution. As the reader may recall, the example in Figure 2 contained the symbol `'OnHold`. The symbol may be used to control the behaviour of an operator or defined to hold a value independent of any operator (query-wide or even system-wide). To resolve symbols to values, we implement two distinct operators: the *symbolic substitution operator* resolves symbol names to expressions using a global hashmap. The *symbolic evaluation operator* traverses an expression and probes the hashmap to find a value for each symbol found in the expression. If the symbol is not found in the hashmap, the symbol itself is returned unevaluated.

5 BOSS - AN IMPLEMENTATION OF HCP

We implement HCP in a system we call BOSS, short for **B**ulk-**O**riented **S**ymbol-**S**tore. BOSS is a DBMS based on the SWM storage and processing models. In this section, we present the implementation principles of BOSS' storage and processing layers.

5.1 Implementing the Storage Layer

Implementing the HCP storage model for a DBMS engine such as BOSS requires two extra features compared to conventional DBMS storage layers: one to store homoiconic data and one to recover the tuple order after applying SWPD.

5.1.1 Storage Backend. When expressions are stored under the SWPD scheme, the expressions' structure must be preserved. To limit the implementation effort, we have implemented it on top of an existing storage backend by integrating Apache Arrow [4]. We extended Apache Arrow's storage layouts with a metadata layer to support homoiconic data. The expressions' arguments are stored as a list of argument arrays and nested expressions (after SWD is applied) as nested arrays. The expression's head is stored as string metadata rather than in a column to avoid redundant information.

5.1.2 Preserving Order. Unlike most RDBMS implementations, which require an ordered relation only for operators such as `TOP`, `OVER` (window aggregations) and `ORDER BY`, BOSS must support the imputation & querying of missing data, which requires an ordered relation for some imputation techniques [31]. This order is usually defined by an attribute, e.g., a timestamp. Unfortunately, the SWP model is inherently unable to guarantee order preservation: the partitioning scheme is designed to re-order the tuples according to the shape of the expressions. To address this problem, we implement an *order-preservation* indexing mechanism to record the tuple order regardless of the number of shape-wise partitions. To determine the most efficient method to iterate over the ordered shape-wise partitions, we evaluated two approaches: (i) a global index; (ii) multiple partitioned indexes.

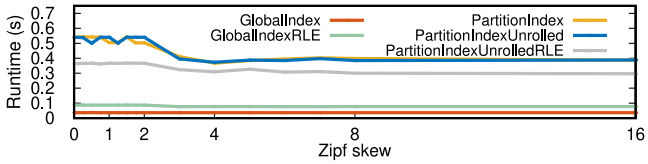

Figure 7: Benchmarking Row Order-Preservation Indexes

In the global index approach, a single index stores a reference to each tuple's partition and its offset from the beginning of that partition. Tuples are retrieved by performing a lookup in the global index followed by a lookup in the respective partition. The second approach requires one index per partition, explicitly storing every tuple's position in the global order. Both approaches are amenable to Run-Length-Encoding (RLE).

To assess the performance of each indexing method, we conduct a performance analysis with an experiment where we unwrap four million 32-bit integer values, wrapped in four different expression shapes, and sum them up. We implemented both approaches with and without additional optimizations: (1) GlobalIndex without RLE; (2) CompressedGlobalIndex using RLE; (3) PartitionIndex without optimization; (4) PartitionIndexUnrolled (optimized by assuming that the next tuple is located in the same partition as the current); and (5) PartitionIndexUnrolled using RLE. This method simulates data in which some values are homoiconic while most are not (a higher skew value indicates fewer expression values).

As shown in Figure 7, the GlobalIndex outperforms the PartialIndexes variations by at least 8×. RLE-compressionIndex yields at least 3× lower memory footprint (depending on the data distribution between the partitions), but the runtime is 2× worse. In addition to the results in Figure 7, we studied a wide range of configurations (e.g., more partitions and wider types) and found similar results. We found that merging partitions into a global order in the PartialIndexes approach incurred many stall cycles due to the microarchitectural complexity of the code (specifically, branch mispredictions). The GlobalIndex, while requiring more bandwidth, allows merging tuples using a simple gather (i.e., a native CPU instruction). Considering the superior performance of the GlobalIndex approach, we use this approach in the system implementation. Due to the memory footprint [4] and the execution cost [5] of initializing one index per row, only the cases listed above require ordering. This is implemented in BOSS as an opt-in: the user explicitly turns the option on before executing a query.

5.2 Implementing the Processing Layer

The HCP processing model requires a trade-off between the efficiency of a typical DBMS and the flexibility and expressivity to

[4] the cost of adding three int32 attributes: global index, partition index, local index
[5] 5x to 50x on typical analytic queries (i.e., TPC-H SF1)

```
template <typename Func> class BinaryOp : public Operator {
  template <typename T1, typename T2>
  void consume(T1 lhs, T2 rhs) {pushUp(Func(lhs, rhs));}
};
Register("Plus", BinaryOp<plus>());
```

Figure 8: Example Implementation of Binary Operators

support the evaluation of homoiconic data. Let us first discuss how the operator API is designed to allow straightforward support for new operators before we describe how homoiconic data is efficiently evaluated during query evaluation.

5.2.1 Operator Implementation and Execution. To allow easy extension with new operators, BOSS builds on statically compiled operators that are implemented using (compile-time-instantiated) C++ templates. When the system starts, pointers to these (type-specific) functions are inserted into an *Operator Registry*: a hashtable whose keys are the shape of the operators' expressions. This allows efficient dispatching of statically typed inputs to operators without the need for advanced techniques such as JIT-compilation. For example, the code in Figure 8 is compiled for each of the sixteen combinations of argument types (e.g., single integer, floating-point value or arrays of those). These statically-compiled implementations are inserted into the hashtable at runtime during the engine's initialization. During the evaluation of the expression (`Plus 1 2`), BOSS calculates the hash for the shape $\langle Plus, Int, Int \rangle$, probes the operator registry, retrieves the matching operator implementation and calls `consume` with the two integer elements as arguments.

5.2.2 Relational Operators. Other than being type-generic, BOSS's operators are in line with state-of-the-art vectorized engines: a *Select* operator that generates a bitset; a *Project* operator that evaluates s-expressions over tuples; a *Sort By* operator that sorts and merges partitions; hash-based *Group By* and hash-based *Join* operators.

5.2.3 Query Evaluation. Unlike persistent expressions, query expressions are not decomposed using SWPD: they are single expressions (i.e., not part of a homoiconic collection) and are depth-first processed through the evaluation pipeline. By convention, the first argument of operators constitutes their input and is eagerly evaluated: table symbols are substituted with the matching partitions and sub-expressions are recursively evaluated. The other arguments (e.g., the predicate of a selection) are passed unevaluated to the operator. To illustrate, consider the evaluation of the query in Figure 3. First, the `Group` operator is evaluated. Its arguments are a `Select` sub-expression and a `'Count` symbol. Starting with the first argument, BOSS recurses into the `Select`. The `Select` operator's arguments are a `'LINEITEM` table symbol and a `Where` sub-expression (the predicate). Because no `Select` operator takes a symbol as the first argument, the first argument is evaluated: the symbol is searched in the table registry and substituted with the matching partitions. The second argument, however, is passed unevaluated to the `Select` operator since an implementation taking such arguments exists: the operator is called for each input partition and iteratively passes the result (a partition) as the first argument to the `Group` operator. Similarly, BOSS finds a `Group` operator that takes a partition as first argument and an unevaluated `'Count` as second argument, calls the implementation for

```
(Group (Table (Schema 'KEY 'SHIPDATE 'DISCOUNT 'TAX)
              (Tuple   1  "96-03-13" (Mean)   .10)
              (Tuple   2  "96-04-12"  .04     .08)) 'Count)
```

Figure 9: Partially-Evaluated Expression as a Query's Result

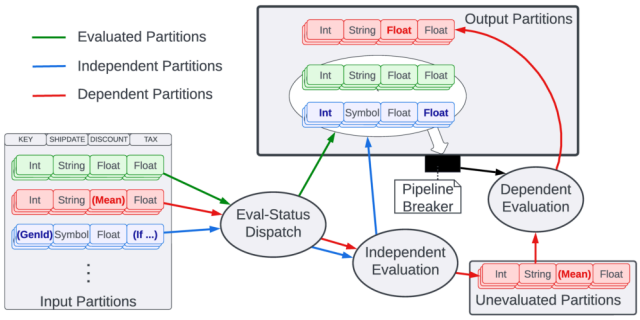

Figure 10: Data Flow of the Evaluate Operator

evaluation and returns the output partitions. If a `Group` with the matching argument types was still not found in the registry, BOSS would have returned instead a partially evaluated expression as the query's output, shown in Figure 9. This case illustrates one of the advantages of homoiconicity: even the query's result can contain unevaluated expressions that could, e.g., encode instructions for visualization or even signal different kinds of errors to the user. Most homoiconic languages (e.g., Mathematica, Racket or Julia) signal errors by returning the unevaluated input expression wrapped in an error expression. In future work, the usefulness of that technique in signalling erroneous queries could be studied.

5.2.4 Evaluate Operator. Thanks to SWPD, all operators discussed so far are evaluated with insignificant overhead because expressions are evaluated without referencing other tuples. We call this type of evaluation *Independent Evaluation*. However, many scenarios, including advanced imputation methods such as (`Mean`), data from other tuples: these must be buffered before being used (the mean, e.g., requires buffering *all* known values in the column). We call this evaluation *Dependent Evaluation*, and we implement it using a dedicated `Evaluate` operator.

Figure 10 illustrates how "independent" expressions like (`GenID`) and "dependent" expressions like (`Mean`) are evaluated in `Evaluate`: during the first phase, called *Eval-Status Dispatch*, input partitions are dispatched depending on their evaluation status (evaluated, dependent or independent). *Evaluated partitions*, i.e., the green ones containing no expressions, are directly pushed to the output (and buffered as input for the next phase). The partition containing (`Mean`), in red, and the one containing (`GenID`), in blue, are passed to *Independent Evaluation*. During this phase, the operator attempts to evaluate expressions without cross-tuple information: an expression such as (`GenID`) is returned evaluated (effectively generating unique integers for all values in the column) whereas (`Mean`), which expects arguments for its evaluation, stays unevaluated. *Independent partitions*, such as the blue one containing (`GenID`), once evaluated, are pushed to the output and buffered. *Dependent partitions*, i.e., containing at least one dependent expression, such as the

```
(Group (Select (Evaluate 'LINEITEM)
           (Where (> 'TAX .07))) 'Count)
```

Figure 11: Example of the Evaluate Operator Placement.

red one containing (Mean), stay unevaluated in this phase as they lack their arguments. In the last step, the remaining unevaluated partitions are passed for *Dependent Evaluation*. This phase evaluates and emits the expressions in all remaining partitions.

To specify at which specific step of the pipeline the partitions are evaluated, the Evaluate operator is explicitly placed in the query expression as shown in Figure 11. To decide where to place it, a simple heuristics is applied: the Evaluate operator is initially just next to the table scan and pushed up the pipeline as far as there is no downstream selection predicate, projected expression, aggregations, grouping, join or sort on columns that contain missing values.

This approach to implementing Evaluate allows the efficient evaluation of dependent expressions, which will be used to implement advanced data imputation operators as detailed next.

5.3 Pay-As-You-Go Imputation in BOSS

Imputation is a natural fit for the "Homoiconic Collection Processing" model. Where conventional imputation systems require the extension of the system with new imputation methods (from scratch and usually at column-granularity), HCP allows users to express such methods at value-granularity, when they enter the data. The system manages the efficient evaluation of the imputation code.

In Figure 2, e.g., the missing values in the TAX column are substituted by the expression (If (> 'SHIPDATE "96-06-01") $x $y). Since the values for $x and $y are stored in dedicated partitions and bulk-evaluated, their evaluation incurs no measurable runtime overhead. In addition, users can store compositions of imputation methods by nesting operators. By composing existing imputation operators, a user effectively defines a new imputation method without requiring any extension of the system or evaluation overhead.

To showcase the expressiveness of the HCP model, we implement four established approaches to data imputation in BOSS.

Approximate Mean [9]. This replaces missing values by the mean of the column's known values. The mean value is computed and memoized by the Evaluate operator and has a near-constant cost.

Hot Deck [9]. This replaces the missing values of a column with random known values from the same column. Where existing implementations [9] require multiple accesses to find a known value, SWD allows to select one in a single access.

Gradient Boosted Trees [11, 28, 41]. As a representative of a learned imputation strategy, this is implemented with a tree ensemble model for regression. This algorithm creates new submodels (i.e., learners) from constituent models' residuals/errors and combines their prediction to improve accuracy and reduce variance. In BOSS, we use the tree-based models provided by XGBoost [11] and accelerate the training runtime with an approximate greedy algorithm using histograms. SWD allows BOSS to access evaluated partitions without additional overhead. The trained model is memoized and reused for subsequent calls to the operator.

Interpolation. This replaces missing values by linearly interpolating the previous and the next known values. These values are retrieved using the GlobalIndex described in Section 5.1.2. SWD enables highly efficient interpolation using batch-wise evaluation.

6 EVALUATION

BOSS supports novel features compared to typical DBMSs: homoiconic storage and evaluation, dynamic typing (due to handling homoiconicity) and imputation operators. In this section, we evaluate BOSS's execution performance when using or not these features.

We start in the evaluation by comparing BOSS with state-of-the-art DBMSs for analytics of non-homoiconic data (6.2) and Wolfram Mathematica and Racket as symbolic data science systems (6.3). Then, we demonstrate the efficiency of BOSS for various imputation methods compared to state-of-the-art approaches (6.4). Finally, we perform micro-benchmarks to investigate how the number of partitions affects the performance (6.5), evaluate its pay-as-you-go properties, i.e., the impact of the number of missing values (6.6) and the benefits of the optimizations we implemented (6.7).

6.1 Experimental Setup

Systems. To assess the runtime performance, we compare BOSS to MonetDB v11.31 [57]; DuckDB v0.5.1 [42]; a commercial RDBMS engine configured as an in-memory column-store; ImputeDB [9]; Mimir Lenses [56]; Wolfram Mathematica v12.3.1 [54] and Racket v6.11 [20]. For the last two, we implemented custom compilers from SQL queries using a data-centric approach in line with the state of the art [36]. Since multi-threaded execution is orthogonal to this research and not yet implemented in BOSS, we configure DuckDB, MonetDB and the commercial engine to run either single-threaded (ST) or multi-threaded (MT) and provide both cases in the results. To compare BOSS and ImputeDB with the exact same imputation operators and configuration, we set ImputeDB's query plan trade-off parameter α to 0 to prioritize quality over performance, ensuring that ImputeDB heuristics do not drop missing values, and, instead consistently executes the imputation as BOSS does.

Workloads. We use the TPC-H benchmark [16] with the scale factor (SF) ranging from 0.001 to 100 (i.e., 1 MB to 100 GB). We followed the established practice [30] to select the five queries that capture complex SQL operations and cover the benchmark's choke points [8]: Q1 for arithmetic and aggregation, Q6 for selective filters, Q3 and Q9 for join processing and Q18 for high-cardinality aggregation (the remaining 17 queries behave similarly with Q3, Q9 or Q18). To benchmark imputation in Section 6.4, we use datasets from CDC [10], FCC [22] and ACS [9]. Following previous work [9], we pre-process the datasets with missing values in most columns, ranging from 0 to 97.89% depending on the attributes, and evaluate the same queries.

Hardware. All experiments are performed on a server with two Intel Xeon Silver 4114 2.20 GHz CPUs, each with 10 physical cores, a 14 MB LLC cache, and 196 GB of memory. We use Ubuntu 18.04 with Linux kernel 4.15.0-167 and compile all code with Clang 11 using the compiler flags `-O3 -mavx2`.

Query Plans. BOSS has no query optimizer yet. To factor out the effects of query optimization, we apply the same plans in DuckDB,

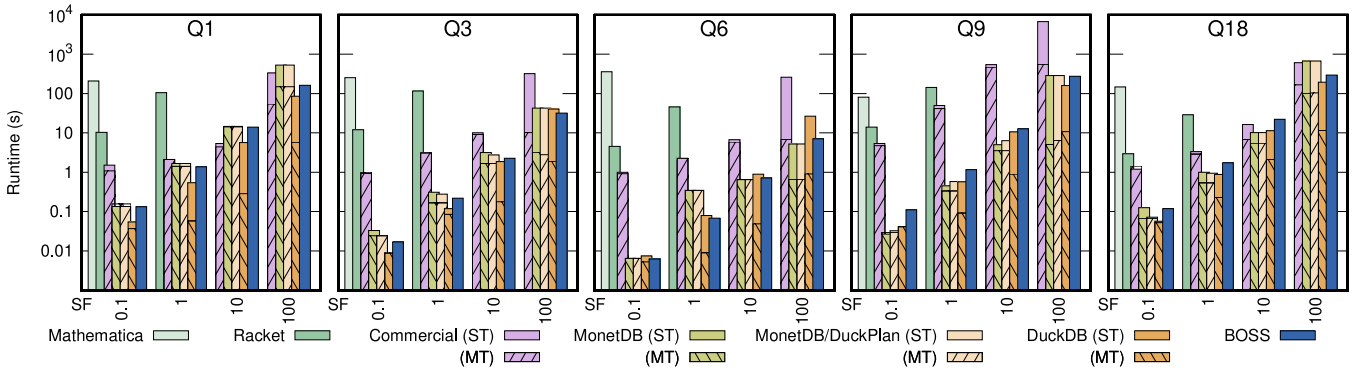

Figure 12: Runtime for TPC-H without Imputation (SF: scale factor)

BOSS and MonetDB (to the extent possible). Since only DuckDB implements join-order-optimization, we use these plans. While this does not assess the best possible performance of each system, it allows us to compare processing performance with other factors equal. For MonetDB, we also run the experiments and report the results for the authors' provided join orders [35] for reference.

6.2 Core DBMS Performance

This experiment assesses the processing overhead compared to state-of-the-art DBMS implementations for introducing homoiconic collections into the system. There are two potential causes of overhead: first, the support for dynamically typed tuples requires column-type dispatch logic per partition in the relational operators' implementation; second, the cost of evaluating the expression per partition. We compare the performance of BOSS with MonetDB and DuckDB, two highly-tuned in-memory DBMSs, not in an attempt to outperform them but as references to measure the overhead in BOSS. We evaluate the selected TPC-H queries discussed in Section 6.1, without missing values.

The results in Figure 12 show that BOSS has competitive performance, placed overall just between single-threaded MonetDB and DuckDB. BOSS outperforms MonetDB and DuckDB for the queries Q3 (with SF 100) and Q6 (except SF 100), but is outperformed by DuckDB for Q1 and Q18, MonetDB for Q6 SF 100 and by MonetDB and DuckDB for Q9. However, BOSS is never more than 2x slower than the other systems. This slowdown can be explained by BOSS not yet implementing some of the optimizations of the other systems (e.g. vectorized aggregation, vectorized hashing and, in general, cache-conscious hash implementation for joins and grouping aggregations). We, therefore, conclude that the proposed approach in BOSS causes no significant overhead that implementation efforts, in line with other DBMSs, could not reduce.

To study the reasons behind the different performance of the systems, we measure the execution time per operator for the five TPC-H queries with SF 10. For BOSS, we instrumented the code and profiled the execution with Intel VTune v2023 [26] to isolate the function calls and measure each operator execution relative to the total query execution time. For MonetDB and DuckDB, we prefixed the query with `TRACE` and `EXPLAIN ANALYZE`, respectively.

Figure 13 show that, in general, BOSS spends more time (relatively) on grouping than other systems (except MonetDB on Q18,

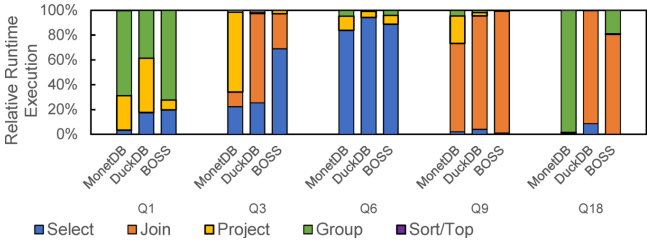

Figure 13: Breakdown of the Relative Runtime per Operator

which exploits the existing group hash to reduce significantly the join execution time). The queries for which BOSS is outperformed by the baselines are dominated by grouping (Q1) and large joins (Q9 and Q18). For Q3, which has smaller tables on the build side of the joins, BOSS spent relatively less time on the join execution: it is less impacted by the cost of building. Q6 is dominated by the selection operator for all systems, which explains why BOSS outperforms the baselines: the comparison operators are more efficiently implemented in BOSS due to better use of SIMD instructions.

6.3 Symbolic Data Science System Performance

To assess the performance advantage of BOSS compared with other homoiconic runtimes, we evaluate BOSS's execution runtime against state-of-the-art symbolic execution engines: Wolfram Mathematica and Racket. Mathematica combines a specialized symbolic execution kernel optimized over decades and an extensive library of data science operators. Racket is a LISP-expression interpreter with efficient runtime. To allow the competitors to optimize queries, we load the dataset into their process memory, cross-compile queries into their respective languages using hyper-style translation [33] and evaluate them using their respective APIs.

Figure 12 illustrates the performance advantage of BOSS over Mathematica and Racket for the TPC-H queries. BOSS outperforms Mathematica by three to five orders of magnitude and Racket by two to three orders of magnitude for small datasets (a maximum SF of 0.1 for Mathematica and 1.0 for Racket since the queries for larger datasets do not finish in time). These results illustrate that, through Shape-Wise Microbatching, BOSS can effectively enable scalable symbolic data science in modern data management systems.

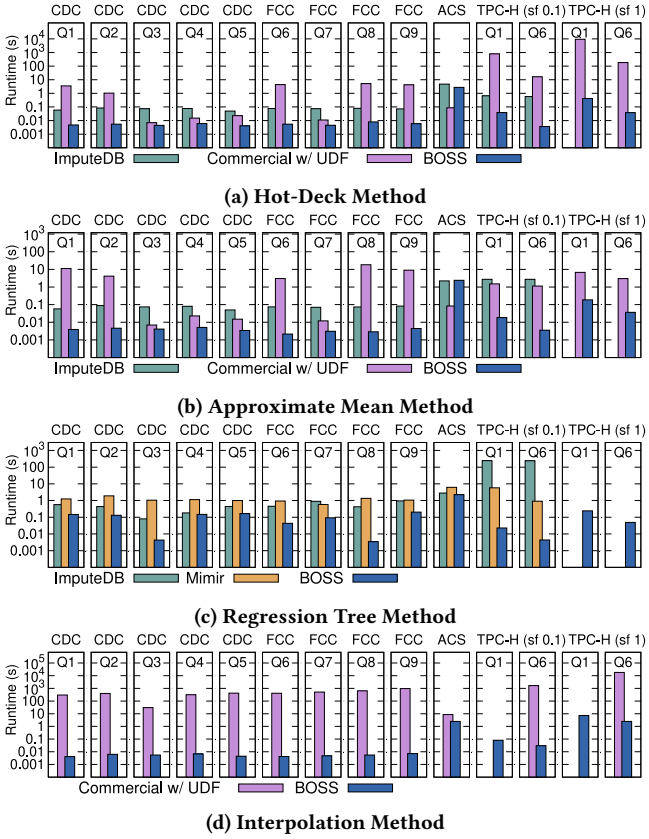

Figure 14: Runtime for Queries with Missing Data Imputation

6.4 Data Imputation System Performance

To evaluate the efficiency of BOSS for missing value imputation, we compare its runtime against three systems: ImputeDB, an experimental system implementing imputation operators in-kernel, Mimir Lenses [56], implemented on top of Apache Spark and thus expected to handle analytics workloads reasonably well, and a classic RDBMS approach, i.e., with the imputation operators implemented as UDFs in a commercial RDBMS engine (one that is known for good UDF support). In this engine, UDFs implementation is very similar to stored procedures. The experiment evaluates all queries for the CDC, FCC and ACS datasets following previous work [9]. To assess scalability, we further evaluate simplified versions of TPC-H Q1 & Q6: as ImputeDB does not support all operations in the original TPC-H queries and handles only integer values, we remove the multi-attribute aggregations and `ORDER BY` clauses and transform all values in the `LINEITEM` table (i.e., strings, floating points, dates) to integers. To generate TPC-H datasets with varying missing values, we replace with NULL 10% randomly selected values from the `DISCOUNT` column. We evaluate four imputation methods presented in Section 5.3: Hot-Deck, Approximate Mean, Regression Tree and Interpolation. Regression Tree is implemented using XGBoost [11] in BOSS, and other third-party libraries in ImputeDB [9] and Mimir [56]. Mimir provides only an implementation for the Regression Tree, whereas we omitted this method for the UDF approach as this cannot reasonably be implemented

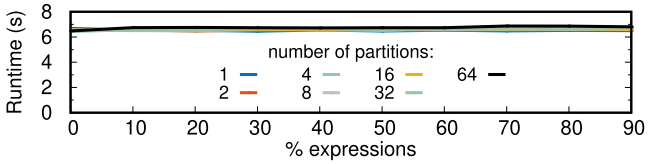

Figure 15: Overhead while Increasing Partition Count

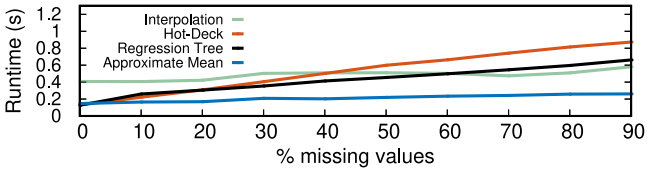

Figure 16: Pay-As-You-Go while Increasing Missing Values

with UDFs, and there is no reason to expect the implementation to be more efficient than other imputation methods. Nor Mimir or ImputedDB implements an Interpolation method.

As illustrated in Figure 14, for the first two methods, BOSS outperforms ImputeDB by one or two order of magnitude on all queries except for ACS (due to many imputed columns for a small dataset). BOSS's performance advantage is due to shape-wise partitioning: it allows BOSS to access clean data without per-value branching (e.g., for computing statistics and for the imputation itself). For the regression trees method, BOSS achieves at least one order of magnitude better performance than Mimir Lenses and several orders of magnitude compared to ImputeDB due to shape-wise partitioning (enabling compiler optimization such as the bulk evaluation of predictions). BOSS outperforms the two UDF implementations by up to several orders of magnitude: the competitor DBMS fails to optimize the query plan effectively due to the increased complexity of a query plan containing such a UDF. The missing bars for the UDF approach, ImputedDB and Mimir Lenses in Figure 14 indicate that the queries could not be executed in time or without running out of memory. The two last columns showing the result for TPC-H at SF 1 demonstrate that only BOSS and the UDF approach can scale to a dataset larger than SF 0.1. However, using the UDF approach, it is not straightforward to implement complex imputations such as the regression tree. Only BOSS achieves both scalability and flexibility.

6.5 Shape-Wise Partitioning Overhead

Since the expression evaluation overhead in BOSS scales with the number of partitions, we assess if there is a noticeable performance degradation when increasing the number of partitions. We evaluate a maximum of eight distinct partitions (since each indicates a different imputation method, this scale is appropriate). We evaluate the execution time using the total time of running the subset of five queries from the TPC-H benchmark with SF 1.0. We replace 0% to 90% of randomly selected values from the `DISCOUNT` column with one to 64 distinct, randomly selected, no-op expressions (so the cost of imputation does not bias the evaluation).

In Figure 15, we observe that BOSS exhibits robust performance independent of the number of partitions and no noticeable overhead when increasing the number of expressions.

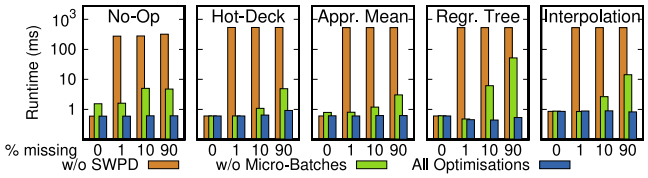

Figure 17: Runtime with Disabled BOSS Optimizations

6.6 Expression Evaluation Overhead

To assess the degree to which BOSS achieves the PAYG objective for imputation, we replace 0% to 90% of randomly selected values from the `DISCOUNT` column from TPC-H at SF 0.1 and impute the values using each of the four imputation methods. To maximize the impact of the imputation on overall performance, we perform imputation on the base table (i.e., before selection). As the execution time of the rest of the query is, therefore, negligible, we execute only Q1.

The results are shown in Figure 16. For Approximate Mean, the time spent to compute the known values is constant because only one calculated value replaces all the missing values. The other three approaches are more expensive due to random memory access for Hot-Deck and Interpolation and mainly the cost for inference for Regression Tree. However, all methods show that the cost increases gradually with the fraction of missing values to evaluate, confirming the PAYG property of the approach.

6.7 Ablation Study

To assess the performance benefits of the proposed optimizations, we modified BOSS to disable them selectively: without SWP (i.e., all column values are expressions if the column has missing values) and without micro-batching (i.e., one unique partition per expression type). We replace 0% to 90% of randomly selected values from the `DISCOUNT` column with no-op expressions (randomly selected among eight unique expressions to simulate using various imputation strategies) in addition to each of the four imputation operators. We evaluate the execution time as the total time running the five TPC-H queries at SF 0.1. Larger SFs do not finish without the optimizations.

The results in Figure 17 show that SWP significantly improves the performance of BOSS. Without this optimization, the execution is 400x slower and the PAYG property is no longer verified: the runtime is constant from 10% to 90% of missing values. Every column value (including non-missing values) suffers the interpretation cost of expression evaluation as soon as one missing value is present in the column. The Micro-batching optimization affects the performance but only to a lesser extent: without this optimization, the execution is 6x to 100x slower due to not ensuring CPU cache efficiency.

7 RELATED WORK

Data Analytics This domain has been extensively researched [19, 30, 38, 39]. Our processing model is based on MonetDB/X100 [57] but requires significant changes to store and evaluate expressions.
Database Extension Systems Most DBMS engines implement UDFs to support custom operators. Unfortunately, UDFs come with significant execution overhead [44]. JIT-compilation can improve UDF execution performance [17, 47, 49] within limits. Extending these to support homoiconic imputation (using custom datatypes to represent missing values and UDFs to interpret them) is conceivable. However, as the custom datatypes would not only have to be dynamic themselves but also force all other values in the column to be dynamically typed, such a solution would suffer from interpretation overhead similar to that of Racket and Mathematica. They could adopt the proposed SWM model to address that.
Storing and Executing Expressions Oracle DBMS allows storing filter expressions [23, 55] in the database (for indexing). In probabilistic databases, expressions are stored for confidence and lineage information [1, 29]. Similar to BOSS, Mimir Lenses [56] allows storing missing values in views rather than stored in the database. M-tables [51] introduces a generalized representation system for missing data but ignores its performance aspects. "Fields of type procedure" have been proposed for Postgres [50] to embed code into column data. We could not find an implementation of the concept in Postgres or any other DBMS (the commercial DBMS we use explicitly prevents the interpretation of code stored in tuples). In addition to the focus on efficiency, our work generalizes all these.
Document Databases Trees/documents [13] could be used to represent expressions. However, document databases differ from homoiconic databases in that documents cannot be evaluated. Further, their storage model is optimized for insert and lookup performance and is highly inefficient for analytics. A homoiconic database is a strict generalization of a document store, allowing a more flexible interpretation of the data for the user.

8 CONCLUSION AND FUTURE WORK

To maintain their usefulness in the face of increasingly varied workloads, DBMSs need to overcome the rigidity of their data and processing models without abandoning their performance advantages. To achieve this, we proposed a new logical data model: *Homoiconic Collection Processing (HCP)*. Implemented in a novel system called BOSS, HCP provides unprecedented extensibility and competitive performance through a novel processing model called *Shape-Wise Microbatching (SWM)*. We demonstrated that SWM delivers the performance of a modern in-memory DBMS and the extensibility of a homoiconic programming language. To demonstrate the extensibility, we implemented data imputation that executes at least an order of magnitude faster than existing imputation-capable DBMSs.

In the future, we see applications for HCP in areas as diverse as generative AI, data visualization, data integration, distributed processing, and even distributed consensus. We also envision applications that require symbolic reasoning, including DBMS-internal applications like query optimization. In addition, we see opportunities to leverage the partial evaluation of the queries to simplify and extend existing DBMS designs. Finally, there are challenges in the implementation of HCP that do not fit the scope of this paper. Implementing query result memoization could benefit performance but is out of scope. The same goes for high-performance transactions: for now, data is bulk-loaded/updated in a "stop-the-world"-manner.

ACKNOWLEDGMENTS

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/W001012/1].

REFERENCES

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha Nabar, Tomoe Sugihara, and Jennifer Widom. 2006. Trio: A System for Data, Uncertainty, and Lineage. *Proceedings of the VLDB Endowment* (2006), 1151–1154. https://doi.org/10.5555/1182635.1164231

[2] Apache. 2023. Open Office Calc. Retrieved 2024-01-22 from https://www.openoffice.org/product/calc.html

[3] Apple. 2023. Apple Numbers. Retrieved 2024-01-22 from https://www.apple.com/numbers/

[4] Apache Arrow. 2023. Retrieved 2023-02-24 from https://arrow.apache.org

[5] Mangesh Bendre, Vipul Venkataraman, Xinyan Zhou, Kevin Chang, and Aditya Parameswaran. 2018. Towards a Holistic Integration of Spreadsheets with Databases: A Scalable Storage Engine for Presentational Data Management. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 113–124.

[6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2015. Julia: A Fresh Approach to Numerical Computing. arXiv:1411.1607 [cs] Retrieved 2024-05-31T17:23:06Z from http://arxiv.org/abs/1411.1607

[7] Peter Boncz and M. L Kersten. 2002. *Monet: A next-Generation DBMS Kernel for Query-Intensive Applications.* Ph.D. Dissertation. Universiteit van Amsterdam.

[8] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 61–76.

[9] José Cambronero, John K. Feser, Micah J. Smith, and Samuel Madden. 2017. Query Optimization for Dynamic Imputation. *Proceedings of the VLDB Endowment* 10, 11 (Aug. 2017), 1310–1321. https://doi.org/10.14778/3137628.3137641

[10] Center for Disease Control. 2016. National Health and Nutrition Examination Survey (2013-2014). Retrieved 2023-02-24 from https://wwwn.cdc.gov/nchs/nhanes/ContinuousNhanes/Default.aspx?BeginYear=2013

[11] 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.

[12] 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.

[13] Chris Clifton, Hector Garcia-Molina, and Robert Hagmann. 1988. The Design of a Document Database. In *Proceedings of the ACM Conference on Document Processing Systems*.

[14] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. 2007. Improving Data Quality: Consistency and Accuracy. *(Proc. VLDB Endow.)*, Vol. 7. 315–326.

[15] George P. Copeland and Setrag Khoshafian. 1985. A Decomposition Storage Model. *Proceedings of the 1985 ACM SIGMOD international conference on management of data* (1985).

[16] The Transaction Processing Council. 2013. TPC-H Benchmark (Revision 2.16.0). Retrieved 2023-02-24 from http://www.tpc.org/tpch/

[17] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Ugur Cetintemel, and Stan Zdonik. 2014. Tupleware: Redefining Modern Analytics. *arXiv:1406.6667 [cs]* (July 2014). arXiv:1406.6667 [cs]

[18] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *CIDR*.

[19] Frans Faerber, Alfons Kemper, Per-Åke Larson, Justin Levandoski, Tjomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends® in Databases* 8, 1-2 (2017), 1–130. https://doi.org/10.1561/1900000058

[20] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Tim Fischer, Denis Hirn, and Torsten Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *Proceedings of the 2022 International Conference on Management of Data*. ACM, Philadelphia PA USA, 2389–2392. https://doi.org/10.1145/3514221.3520175

[22] FreeCodeCamp. 2016. New Coder Survey. Retrieved 2023-02-24 from https://www.kaggle.com/freecodecamp/2016-new-coder-survey-

[23] D. Gawlick, D. Lenkov, A. Yalamanchi, and L. Chernobrod. 2004. Applications for Expression Data in Relational Database Systems. In *Proceedings. 20th International Conference on Data Engineering*. IEEE Comput. Soc, Boston, MA, USA, 609–620. https://doi.org/10.1109/ICDE.2004.1320031

[24] Google. 2023. Google Sheets. Retrieved 2024-01-22 from https://www.google.com/sheets/about/

[25] Georg Gottlob and Roberto V Zicari. 1988. Closed World Databases Opened through Null Values.. In *VLDB*, Vol. 88. 50–61.

[26] Intel. 2023. VTune Profiler. Retrieved 2023-02-24 from https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[27] Mohamed Ismail and G. Edward Suh. 2018. Quantitative Overhead Analysis for Python. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Raleigh, NC, 36–47. https://doi.org/10.1109/IISWC.2018.8573512

[28] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A Highly Efficient Gradient Boosting Decision Tree. *Advances in neural information processing systems* 30 (2017).

[29] Oliver Kennedy and Christoph Koch. 2010. PIP: A Database System for Great and Small Expectations. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, Long Beach, CA, USA, 157–168. https://doi.org/10.1109/ICDE.2010.5447879

[30] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask. *Proceedings of the VLDB Endowment* 11, 13 (Sept. 2018), 2209–2222. https://doi.org/10.14778/3275366.3275370

[31] 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. *Proceedings of the VLDB Endowment* 13, 5 (Jan. 2020), 768–782. https://doi.org/10.14778/3377369.3377383

[32] Yiming Lin and Sharad Mehrotra. 2023. ZIP: Lazy Imputation during Query Processing. *Proceedings of the VLDB Endowment* 17, 1 (Sept. 2023), 28–40. https://doi.org/10.14778/3617838.3617841

[33] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

[34] Microsoft. 2024. Excel. Retrieved 2024-01-22 from https://www.microsoft.com/microsoft-365/excel

[35] MonetDB. 2024. TPC-H Scripts for MonetDB. Retrieved 2024-07-13 from https://github.com/MonetDBSolutions/tpch-scripts

[36] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[37] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-Learn: Machine Learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[38] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. 2013. CPU and Cache Efficient Management of Memory-Resident Databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Brisbane, QLD, 14–25. https://doi.org/10.1109/ICDE.2013.6544810

[39] Holger Pirk, Oscar Moll, Matei Zaharia, and Sam Madden. 2016. Voodoo-a Vector Algebra for Portable Database Performance on Modern Hardware. *Proceedings of the VLDB Endowment* 9, 14 (2016), 1707–1718.

[40] PostgreSQL. 2023. What Is PostgreSQL? Retrieved 2023-02-24 from https://www.postgresql.org/about/

[41] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. 2018. CatBoost: Unbiased Boosting with Categorical Features. *Advances in neural information processing systems* 31 (2018).

[42] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science Towards Embedded Analytics. (2020).

[43] Karthik Ramachandra and Kwanghyun Park. 2019. BlackMagic: Automatic Inlining of Scalar UDFs into SQL Queries with Froid. *Proceedings of the VLDB Endowment* 12, 12 (Aug. 2019), 1810–1813. https://doi.org/10.14778/3352063.3352072

[44] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of Imperative Programs in a Relational Database. *Proceedings of the VLDB Endowment* 11, 4 (Dec. 2017), 432–444. https://doi.org/10.1145/3186728.3164140

[45] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proceedings of the VLDB Endowment* 10, 11 (Aug. 2017), 1190–1201. https://doi.org/10.14778/3137628.3137631

[46] El Kindi Rezig, Mourad Ouzzani, Walid G Aref, Ahmed K Elmagarmid, Ahmed R Mahmood, and Michael Stonebraker. 2021. Horizon: Scalable Dependency-Driven Data Cleaning. *Proc. VLDB Endow.* 14, 11 (2021), 2546–2554.

[47] Maximilian E. Schüle, Jakob Huber, Alfons Kemper, and Thomas Neumann. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *32nd International Conference on Scientific and Statistical Database Management*. ACM, Vienna Austria, 1–12. https://doi.org/10.1145/3400903.3400915

[48] Scikit-learn. 2022. Scikit-Learn: Imputation of Missing Values. Retrieved 2023-02-24 from https://scikit-learn.org/stable/modules/impute.html

[49] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. (2022).

[50] Michael Stonebraker and Lawrence A Rowe. 1986. The Design of POSTGRES. *Proceedings of the 1986 ACM SIGMOD international conference on management of data* (1986), 340–355.

[51] Bruhathi Sundarmurthy, Paraschos Koutris, Willis Lang, Jeffrey Naughton, and Val Tannen. 2017. M-Tables: Representing Missing Data. (2017). https://doi.org/10.4230/LIPICS.ICDT.2017.21

[52] Devesh Tiwari and Yan Solihin. 2012. Architectural Characterization and Similarity Analysis of Sunspider and Google's V8 Javascript Benchmarks. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE,

New Brunswick, NJ, USA, 221–232. https://doi.org/10.1109/ISPASS.2012.6189228

[53] Wolfram. 2022. How To Replace or Remove Invalid or Missing Data. Retrieved 2023-02-24 from https://reference.wolfram.com/language/howto/ReplaceOrRemoveInvalidOrMissingData.html

[54] Stephen Wolfram. 1991. *Mathematica: A System for Doing Mathematics by Computer.* Addison Wesley Longman Publishing Co., Inc.

[55] Aravind Yalamanchi, Jagannathan Srinivasan, and Dieter Gawlick. 2003. Managing Expressions as Data in Relational Database Systems. (2003).

[56] Ying Yang, Niccolò Meneghetti, Ronny Fehling, Zhen Hua Liu, and Oliver Kennedy. 2015. Lenses: An on-Demand Approach to ETL. *Proceedings of the VLDB Endowment* 8, 12 (Aug. 2015), 1578–1589. https://doi.org/10.14778/2824032.2824055

[57] Marcin Zukowski, Peter A Boncz, Niels Nes, and Sándor Héman. 2005. MonetDB/X100-A DBMS in the CPU Cache. *IEEE Data Eng. Bull.* 28, 2 (2005), 17–22.