Homoiconicity For End-to-end Machine Learning with BOSS

Hubert Mohr-Daurat[1], Holger Pirk[1]

[1]Imperial College London

Abstract

It is common for machine learning applications to have complex operator pipelines, with several steps before and after the main execution of the core machine learning algorithm. Consequently, data movement between processes has a performance cost for the execution of the application. In this paper, we propose an end-to-end machine learning framework based on the principle of homoiconicity: data and code are represented in a single unified syntax. This approach allows the efficient interpretation of custom operators at the core of a relational database system. We also present the implementation of a data cleaning framework as an initial milestone. This work is a part of a larger research project to implement BOSS, a novel, general-purpose relational database system that stores and processes homoiconic data.

Keywords

relational database, homoiconicity, symbolic expressions, machine learning, data cleaning, data imputation

1. Introduction

The core part of any machine learning (ML) pipeline is the model to be learned during training and executed during inference. It is, thus, not surprising that this is the most extensively researched and optimized aspect of ML applications.

However, the model training and inference is not the only component of the data processing pipeline that plays a role in the application's overall performance. Many applications move large amounts of data as inputs and produce large output datasets. This data movement is costly and, often, principally unnecessary.

In addition, many applications perform internal data movement for various data operations, such as data augmentation or data normalisation. Some of these steps involve the usage of dedicated libraries and even systems.

The complexity of such pipelines with multiple layers increases not only the overhead for data movement but also the engineering cost of implementation and maintenance of the logic to connect them.

Data cleaning is a particularly interesting example of such data movement-intensive components in the case of an ML pipeline. Cleaning the data is necessary because the model learning and the inference process are extremely sensitive to the data quality. Input data is therefore generally cleaned using methods such as imputing missing values and searching and fixing incorrect values (based on common sense, expert knowledge, and user-defined constraints). Algorithms for these tasks are integrated into the pipeline, from simple averages or interpolations to more advanced regression methods.

Consequently, ML pipelines are often complex, delegating these additional data transformations to separate processes that cause significant data movement during their execution; to load data into memory, querying and passing it to the data cleaning component (before or during querying the data), passing it to the core ML component (the learning or the inference) and then again moving the output data to the application layer, which consumes the result.

Database Management Systems (DBMS), where the data lives for any data-heavy processing pipeline, plays a significant role in the data movement problem with a high overhead for transferring the data in and out.

Prior research [1] has investigated methods to reduce the data movement overhead out of the DBMS. With ad-hoc serialisation protocols and efficient compression technologies, the overhead is reduced but not eliminated and can still be significant for large-scale data movement.

Instead of making the data transfer more efficient, an alternative approach is to reduce the cases where data movement is needed, e.g. by executing the various data retrieval and transformation steps in the same process or by using shared memory when independent processes are required. However, data movement can rarely be eliminated due to the additional transformations needed to match the data layout used by the libraries and applications used in each layer.

A more radical solution to this problem is integrating all the layers of data processing steps into a single central system. This approach is much more effective at reducing the data movement overhead.

The DBMS is a natural starting point for such a central system. DBMS are designed to store, retrieve and pro-

BICOD'21: British International Conference on Databases, Mars 28, 2022, London, UK

h.mohr-daurat19@imperial.ac.uk (H. Mohr-Daurat);
hlgr@ic.ac.uk (H. Pirk)

cess a large amount of data efficiently, taking advantage of strategies and technologies studied, developed and improved over several decades.

However, implementing all the desired features in the DBMS requires new data representations, operators and logic, which are not standard in typical DBMS. This is challenging to implement while preserving the advantages of the DBMS (such as the query optimizer).

Implementing the desired features in a database comes with a trade-off. Implementing them as user-defined functions (UDFs) is straightforward but has a high overhead and prevents query optimizations; implementing them at the core of the DBMS backend is more efficient but requires significant effort and expert knowledge.

Instead, we propose implementing a more generic extensible system that is flexible and takes advantage of the DBMS efficiency to manipulate data. To achieve that, the DBMS stores as data the executable code that extends the DBMS functionality. Storing the code as data means that the code is an integral part of the relational representation and can be manipulated by the DBMS kernel. This method is fundamentally different from UDFs and triggers, which exist only in the schema and are be retrieved and run as a black box at query time. With our approach, the DBMS handles this custom logic efficiently, taking advantage of the optimization strategies produced by DBMS research.

Fundamentally, homoiconicity refers to this idea that code can be read, modified and stored like data. The practically most common forms of homoiconic data are symbolic expressions (s-expressions). An s-expression is represented as a (potentially nested) list and can, therefore, be modified as well as evaluated. For example, '1 + 2' would be represented by the list '(Plus 1 2)'. Lisp-like languages use s-expressions to represent all code as s-expressions and allow Turing-complete programming [2]. They even allow the representation of unknown or undefined values in the form of "Symbols". Unfortunately, lisp implementations are ill-suited to support data-intensive applications due to prohibitively high interpretation overhead (see Section 4).

Inspired by these homoiconic languages, we are implementing a DBMS called BOSS for Bulk-Oriented Symbol-Store. This database stores symbolic expressions in the form of symbols and complex expressions. However, in our implementation, the evaluation of the expressions is performed for a large batch of data at once, thus amortizing interpretation overhead.

Applied to the ML domain, native operators are implemented in the database only for the basic blocks of the ML operations (e.g. transformations, activations). Only the logic of the ML algorithm, less generic, is left to be implemented in userspace with symbolic expressions.

The benefits of this approach are:

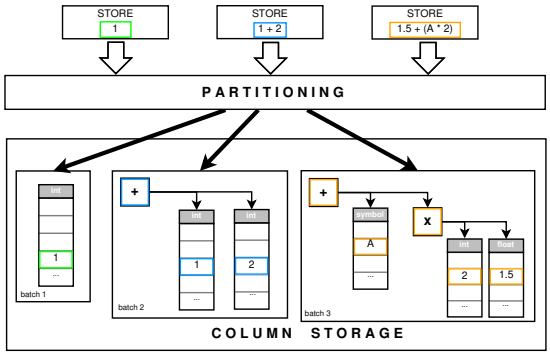

Figure 1: *shape-wise partitioning* mechanism and storage representation example.

- ease for the user to implement new ML algorithms and tools by assembling them from operators using symbolic expression logic with flexibility while keeping native performance
- with the logic as data paradigm, the approach opens new possibilities for optimizations, unified for both the data (as usually handled by the query optimizer) and for the logic (as usually handled by the compiler) by exposing the logic as symbolic data to the database at every level, from query plan to storage layers.
- symbolic expressions are not used only to express the logic of the queries. They are also stored in the database and evaluated efficiently at query time, whose benefits and implementation are detailed in the use case in the next section.

2. Storing and Processing Homoiconic Data

As data cleaning is a data-intensive task, we, naturally, selected the decomposed storage model (DSM) [3] to represent (plain) relational data. On top of DSM data, we implemented the BOSS kernel following the X100-model[4]: BOSS' query processor partitions data into micro-batches to keep intermediate results cache-resident. In addition to plain relational data, however, BOSS needs to process homoiconic expressions that represent computation efficiently. Interpreting such expressions usually incurs significant overhead.

To amortize this overhead, we developed a novel technique called *shape-wise partitioning & decomposition* as shown in Figure 1. First, collections of expressions are (horizontally) partitioned by shape, i.e., all expressions

that only differ in their base type arguments form a partition. Second, the components of the expressions are (recursively) decomposed, thus storing base data in a decomposed form. In Figure 1, e.g., batch 1 stores plain integers, while batch 2 stores expressions that add two integers and batch 3 stores expressions that multiply two integers and add them to a third. As shape-wise partitioning ensures homogeneous partitions, entire partitions can be processed with a minimal number of function calls (one per subexpression). This ensures highly CPU-efficient processing.

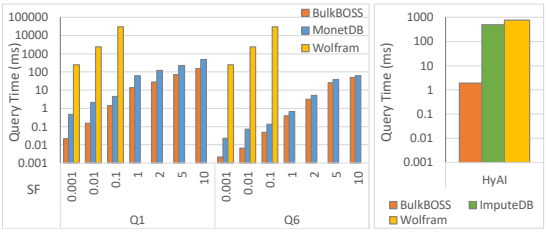

Figure 2: Performance comparison tests for TPC-H Q1/Q6 (relational), and HyAI (imputation). The missing bars for Wolfram are caused by the engine running out of memory.

3. Use case: data imputation

We implemented a data imputation framework in the context of a project called HyAI[1]. The project's overall goal was to optimize the decisions to charge or consume hydrogen batteries using machine learning technologies that model future electricity consumption from the analysis of many different data sources. Because the batteries were intended to be deployed in areas where data transferred from these sources might be unreliable, a robust imputation framework was implemented in the database management system so that the user could define custom and potentially complex imputation logic.

Missing data is commonly handled in database systems by inserting null values and internally defining an arbitrary value as null or using a boolean flag. This method does not allow to specify why the data is missing and to let the data cleaning process decide how the value is imputed. However, there are multiple reasons why data would be missing. Ideally, the user should be allowed to specify through code logic the intent and decide how to handle missing data.

One approach for the user to handle missing values is by calling the imputation method on the fly before inserting the data into the database. However, this method does not allow the use of future data for the imputation. For example, both previous and subsequent values may be needed for the interpolation of time series data.

Alternatively, the user could handle missing values as part of the query logic by detecting the missing values and using nested queries or UDFs. However, this method usually incurs an execution overhead that affects performance and increases the queries' complexity.

With the imputation framework that we implemented, the user stores missing data as complex expressions. The expressions describes the logic to use for the data imputation, e.g. instead of inserting a 'null' value, the user inserts the expression '(PreviousValue() + NextValue()) / 2' to do a simple local average. As presented in the previous section, this approach provides more flexibility to the user and benefits from the optimizations allowed by the bulk evaluation method.

[1] <https://www.h2gopower.com/hyai>

4. Evaluation

All the experiments have run on a PC with an Intel i9-10990K CPU and 32GB RAM DDR4 3600MHz.

To confirm that relational operators in BOSS are performing competitively with state-of-the-art DBMS, we compared the performance of BOSS with MonetDB using the TPC-H queries Q1 and Q6 with various scale factors. The results in Figure 2 shows that BOSS performs on the same magnitude with MonetDB without an impact from having to support homoiconicity in our implementation.

Also shown in Figure 2, we compared the handling of missing data with another implementation, ImputeDB [5], which has similar functionalities. We evaluated a dataset from HyAI project (50K rows) where 50% of the data is missing at random for five columns. Data is imputed by taking a random value from data in the same column. BOSS performs better with a factor of x261. This result shows that the bulk approach to implementing imputation operators is highly efficient. Our method to evaluate expressions efficiently could also benefit more advanced imputation techniques.

In addition, the comparison for both relational operators and missing data evaluation with another backend based on the Wolfram engine shows that our implementation outperforms by several orders of magnitude the commercial homoiconic engine. It confirms that evaluating homoiconic data comes with a high cost for interpreting the expressions, which is solved with the technique we have implemented in BOSS.

5. Related Work

Data cleaning is a critical aspect of many decision-making and analytics applications, particularly in ML-based software. Some solutions have been proposed to automate the process, such as HoloClean [6], an entire system run

side-by-side with the DBMS. Alternative solutions have been proposed to integrate the data cleaning operators as part of the database query, such as ImputeDB as an imputation module for the Data Civilizer framework [7]. Both approaches do not allow the user to make decisions during the data cleaning, except for some initial setup in the case of HoloClean. They differ from our method, which performs data imputation during query but is based on the information provided by the user during data insertion.

Most research on accelerating the ML pipeline and using a unified data management solution focuses on the acceleration of the training step. The approach in [8] has similarities with our work. They propose lambda expressions as generic user-defined operators, which are more efficient than usual UDFs. However, lambda expressions are used only in the queries and do not store code logic in the database. Their focus is primarily on training and inference and not on data cleaning. Their solution also differs from ours since they use JIT-compilation to optimize the evaluation of the expressions.

Tupleware [9] also proposes a similar approach for their general end-to-end analytic solution by analysing the code of UDFs and taking advantage of JIT compilation to find the best pipeline strategy when merging the custom operator code with the query code. The drawback is a warm-up overhead before the workload execution, which is negligible for heavy workloads but can be a problem when a fast response is required [10] (e.g. visualisation, or high-frequency inference).

In addition, there is research work on integrating ML operators as part of database kernel ([11], [12], [13]). This approach is efficient since ML operators are integrated deep inside the database implementation but require investing time to implement ad-hoc code for each needed operator. Similarly, LMFAO [14] propose to represent ML operators as aggregate queries. This method allows interesting optimizations but is limited to operators that can be represented as aggregates. Some research takes the opposite approach and implements the relational operators in a linear algebra kernel [15]. Finally, another approach uses a unified intermediate language for both relational and ML operators ([16], [17]).

6. Conclusion and Future Work

Data movement is a significant performance factor for modern data processing pipelines: the complexity of data processing tasks involving steps such as data cleaning, normalization, model training and visualization already stresses interfaces and will only increase. To address this problem, we developed a new data processing system architecture around the idea of homoiconicity, i.e., the uniform representation of data and code. In this design, task-specific code can be stored and processed with the data, thus providing unprecedented extensibility. In this paper, we have demonstrated the advantages of this design for handling data imputation.

For the future, we plan to extend the system with additional features for data processing pipelines: operators to support additional data cleaning use cases, algebraic operators needed for model training and inference and even domain-specific operators. The system will, further, benefit from relational components such as query optimization, transactional semantics and data visualization.

References

[1] M. Raasveldt, H. Mühleisen, Don't hold my data hostage: A case for client protocol redesign (2017).

[2] J. McCarthy, Recursive functions of symbolic expressions and their computation by machine, Part I, CACM (April '60).

[3] P. Boncz, M. L. Kersten, Monet: A next-Generation DBMS Kernel for Query-Intensive Applications, Ph.D. thesis, Universiteit van Amsterdam, 2002.

[4] M. Zukowski, et al., Monetdb/x100-a dbms in the cpu cache., IEEE Data Eng. Bull. (2005).

[5] J. Cambronero, et al., Query optimization for dynamic imputation, PVLDB (2017).

[6] T. Rekatsinas, X. Chu, I. F. Ilyas, C. Ré, HoloClean: Holistic data repairs with probabilistic inference, PVLDB (2017).

[7] D. Deng, et al., The data civilizer system, in: Cidr, 2017.

[8] M. Schüle, et al., In-database machine learning: Gradient descent and tensor algebra for main memory database systems, BTW (2019).

[9] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, U. Cetintemel, S. Zdonik, Tupleware: Redefining Modern Analytics, arXiv:1406.6667 (2014).

[10] T. Kraska, Northstar: An interactive data science system, PVLDB (2018).

[11] B. De Boe, et al., IntegratedML: Every SQL Developer is a Data Scientist, in: DEEM@SIGMOD, 2020.

[12] K. Karanasos, et al., Extending Relational Query Processing with ML Inference, CIDR (2020).

[13] J. Hellerstein, , et al., The MADlib Analytics Library or MAD Skills, the SQL, arXiv:1208.4165 (2012).

[14] M. Schleich, et al., A Layered Aggregate Engine for Analytics Workloads, in: SIGMOD, , 2019.

[15] L. Chen, A. Kumar, J. Naughton, J. M. Patel, Towards linear algebra over normalized data, PVLDB (2017).

[16] A. Shaikhha, et al., Multi-layer optimizations for end-to-end data analytics, in: CGO@PPoPP, 2020.

[17] S. Palkar, et al., Weld: A Common Runtime for High Performance Data Analytics (2017).