Benchmarking Machine Learning Pipelines in PostgreSQL with TPCx-AI

Leonhard Liu and Patrick Erdelt

June 7, 2024

Benchmarking Machine Learning Pipelines in PostgreSQL with TPCx-AI

Leonhard Liu and Patrick K. Erdelt[0000−0002−3359−2386]

Berliner Hochschule fuer Technik (BHT), Luxemburger Strasse 10, 13353 Berlin, Germany, leonhard.liu@bht-berlin.de

Abstract. Driven by advancements in model capabilities and ease of access, machine learning (ML) and artificial intelligence (AI) are increasingly applied across industry and government sectors. Traditionally, ML training and serving either relies on big external service providers such as AWS or MS Azure, or requires data to be transferred from databases or data lakes to local or cloud environments. Apart from dependencies on external ML frameworks, these type transfers not only introduce significant overhead but also pose risks to data security and data integrity. Integrating these technologies directly within database systems promises significant advantages, particularly for production environments. However, the performance and capability of database systems for various ML scenarios remain unclear. To address these uncertainties, this paper proposes transferring the TPCx-AI benchmark toolkit into PostgreSQL using the MADlib extension. This enables the entire ML pipeline—from data loading and preprocessing to training, scoring, and serving—to be executed within the database system. We present the implementation details and compare its performance with the traditional Python-based approach from the toolkit. The implementation focuses on traditional ML algorithms and does not include Deep Learning techniques. Our evaluation, leveraging the synthetic data generator PDGF and use cases provided by TPCx-AI, offers a comprehensive analysis of the benefits and shortcomings of in-database ML training with PostgreSQL and MADlib. On an aggregated level, it shows a comparable performance between both system for most use cases. While the Python approach excels at model training, PostgreSQL with Apache MADlib demonstrates superior performance in data processing and inference tasks.

Keywords: Database Management Systems · Performance Evaluation · Machine Learning · TPCx-AI · Benchmarking · PostgreSQL · Apache MADlib

1 Introduction

The rapid growth of big data over the past few decades has transformed the landscape of machine learning (ML). Affordable storage, simplified data collection, and ubiquitous sensors have made vast amounts of raw data readily available, enabling the training of highly accurate ML models. However, the preprocessing and transformation of this data into a format suitable for ML algorithms remains a significant challenge, often consuming the majority of the workload for ML engineers and data scientists.

Traditional end-to-end ML pipelines in production environments involve multiple teams and complex hardware integration, driven by the need for retraining due to concept and feature drift. This complexity has motivated the exploration of alternative approaches, such as integrating ML capabilities directly into database systems. Relational databases, with their structured data format, are well-suited for many ML algorithms where features are represented as columns.

PostgreSQL, a widely-used open-source relational database, possess numerous extensions supported by an extensive community of developers. One such extension is Apache MADlib, which provides data-parallel implementations of mathematical, statistical, graph and machine learning methods for structured and unstructured data [11]. To evaluate the effectiveness of in-database ML using PostgreSQL and MADlib, we turn to TPCx-AI, a modern end-to-end benchmark that applies ten widely-used ML algorithms to industry-relevant problems [6]. It leverages the Parallel Data Generator Framework (PDGF) for scalable synthetic data generation, and provides a toolkit for two implementations: one using Python frameworks for datasets with smaller scale factors (SF) and another leveraging Apache Spark for large-scale deployment.

In this paper, we replicate the end-to-end Python benchmark of the TPCx-AI toolkit within the database. This approach eliminates the need for data transfers between systems, reducing overhead and improving data security and integrity.

2 Contribution

This paper demonstrates the successful porting of seven out of ten use cases from the TPCx-AI benchmark onto a PostgreSQL 15 database, leveraging the Apache MADlib library for in-database machine learning. Two out three remaining use cases utilize audio and video. These were not ported due to the unconventional practice to store such formats directly in a database. The final unported use case involves deep learning for regression tasks. Although MADlib supports deep learning through the Python frameworks TensorFlow and Keras, its current capacity is limited to classification problems.

The seven successfully ported use cases encompass a diverse range of algorithms, including clustering, classification, time-series forecasting, singular value decomposition and more. Implementation involves the use of several SQL derivatives and extensions: data preprocessing and manipulation tasks were primarily handled using ANSI SQL, with some PostgreSQL-specific syntax required for certain operations. The MADlib extension was instrumental in facilitating ML training and inference tasks.

For K-means clustering problem (use case 1), we propose the Silhouette score as a qualitative metric. Additionally, we implemented reusable scoring functions, such as the Matthews Correlation Coefficient and Mean Squared Logarithmic Error, to enhance the evaluation capabilities of PostgreSQL and Apache MADlib.

We conducted a comparative analysis between the Python implementation of TPCx-AI and our PostgreSQL implementation. As with the Python implementation, our benchmark runs on a single host. The complete code is available on GitHub[1]. The repository includes Data Definition Language (DDL) scripts for all implementations, a SQL file for executing all steps, and a configuration file for DBMSBenchmarker [10,9]. Additionally, the repository provides scripts for generating Docker images of all components (data generator, driver, loader, and PostgreSQL/MADLib with all libraries installed) to ensure seamless reproduction of the benchmark environment and results.

3 Related work

In-Database Machine Learning (ML) encompasses a wide range of methodologies [14,5]. MADLib exemplifies a system that leverages user-defined (aggregate) functions to incorporate ML functions, particularly for training purposes. In reference [8], the authors describe techniques and experiences of database engine developers, data scientists, IT architects and academics with data-parallel implementations of analytics in a Greenplum cluster of 42 nodes. Based on these experiences, the authors of [11] introduced MADLib in 2012 as an open-source extension for PostgreSQL and Greenplum. MADLib has been utilized in both academic projects and industrial applications. Since its promotion to a top-level project of the Apache Software Foundation in 2017, MADLib has undergone several updates. Recently, MADLib has been evaluated in a setup involving distributed deep learning for classification [19].

TPCx-AI is a relatively new benchmark. Despite this, there are already several studies on it. An introduction to the benchmark and its underlying concepts is provided in [6]. The authors also present results for various scaling factors. TPCx-AI is comparable to MLPerf, as there are some similarities. Both concern the performance of machine learning-related implementations. However, there are also differences, which are closely examined in [15]. This includes aspects such as result review types, licenses, costs, and metrics. The most prominent difference, in our opinion, is that TPCx-AI is an end-to-end benchmark, bringing it closer to production environments. In addition, we are interested in including processes and steps that are not primarily focused on ML training, but take an end-to-end point of view. On the other hand, MLPerf aims to include the latest ML algorithms. In [17], the authors examine the scaling behavior of the benchmark and the toolkit. They provide an introduction to the toolkit and

[1] https://github.com/perdelt/TPCx-AI_in_DB

its configuration parameters. They also analyze a single-node and a multi-node experiment, observing resource consumption for use case 3. In [13], the authors report their first adopter's experiences with a scaling factor of 1,000 in a Spark environment of three nodes. All these studies pertain to settings directly covered by the provided toolkit. However, there is also a result for non-default environments: In [4], the authors perform a translation to NVIDIA Jetson, noting that memory is a crucial aspect. The authors report some necessary changes that had to be made, although it remains a Python-based implementation.

To the best of our knowledge, the only other benchmark for data science applications that compares Anaconda (and Dask, PySpark, and R) with PostgreSQL is Sanzu [18]. It consists of six micro-benchmarks (basic file I/O, data wrangling, descriptive statistics, distribution and inferential statistics, time series, machine learning) and two macro-benchmarks (full use cases, containing linear regression and grouping). The methodology of cataloging and analyzing standard workflow components appears promising. Nevertheless, we selected TPCx-AI due to its incorporation of more complex and versatile end-to-end use cases.

4 Methodology

4.1 Research Questions

The primary goal of this research is to investigate the capabilities of PostgreSQL with Apache MADlib for in-database machine learning, using TPCx-AI. Specifically, we seek to answer the following questions:

1. How does the performance of PostgreSQL with MADlib compare to traditional Python-based ML implementations?
2. What are the strengths and limitations of using PostgreSQL with MADlib for end-to-end ML tasks?
3. Can in-database ML reduce the overhead associated with data transfer?
4. Does TPCx-AI offer a promising guideline for evaluating in-database ML tasks?

4.2 Experimental Design

The core components of our setup are the open-source database PostgreSQL version 15 and Apache MADlib 2.1.0, running on a single machine with the following configuration: Architecture: x86_64, OS: Ubuntu 22.04.4 LTS, CPU: 13th Gen Intel i9-13900K, RAM: 125 GiB.

PDGF was used to generate synthetic data with scale factors 1, 5, 10, and 15 (SF1, SF5, SF10, and SF15). Data loading is handled with Python `Psycopg` library, preprocessing via PostgreSQL, training, and inference via Apache MADlib. Scoring functions are defined in SQL, utilizing ML metrics suggested by TPCx-AI. Key performance metrics such as runtime and output table size, were automatically saved after each benchmark run for further analysis. For evaluation, we use the runtime results of the Power Training Test and the Power Serving Test from TPCx-AI benchmark as baseline for comparison. All Experiments at different scale factors were repeated multiple times and on the same hardware configuration to ensure consistency and comparability.

5 Implementation

5.1 Implementation overview

Hyperparameters from the training functions of the Python implementation were closely matched in each of the MADlib implementations. When certain algorithms were unavailable in MADlib, we substituted them with similar ones. For use case 4, Naive Bayes was implemented purely using SQL. Although Python code can be integrated via the `plpythonu` extension, we opted against it to accurately assess the native capabilities of PostgreSQL and Apache MADlib. Our end-to-end pipeline includes the following steps:

1. **Data generation via PDGF**: The user specifies the size of the dataset with the *scale_factor* parameter.
2. **Data loading and indexing**: An import script loads PDGF-generated CSV and PSV files into the three schemas: train, score, and serve. It also creates indexes to enhance retrieval speed.
3. **Benchmark execution**: This step involves data preprocessing, model training, metric scoring, and model serving for each use case. Figure 1 shows the workflow and interdependencies.

One key difference between our implementation and the one of TPCx-AI lies in the runtime calculation. In TPCx-AI, load test represents the movement of existing raw data generated by PDGF to persistent storage. The reading of the CSV or PSV files in Python, and the preprocessing of the loaded files is an integral part of the Power Training or Power Serving test. Our implementation, on the other hand, offers a more fine-grained analysis of each step in the end-to-end benchmark. We record the preprocessing time for training and serving separately to provide a detailed understanding of the time costs associated with either phase in the pipeline. An end-to-end pipeline on one use case has a code structure as provided below:

```
call uc07_preprocess('train', 'uc07_train_preprocessed');
call uc07_train('uc07_train_preprocessed', 'uc07_model', adjust_params);
call uc07_predict('score','uc07_score_predictions', 'uc07_model');
call uc07_score('uc07_score_predictions', 'uc07_score_results');
call uc07_serve('uc07_serve_results');
```

Each use case utilizes the five standard procedures *Preprocess*, *Train*, *Predict*, *Score*, and *Serve*. Their functionalities are:

1. **Preprocess**: Selects and joins the appropriate table(s) based on the specified schema; prepares the data for training or inference.

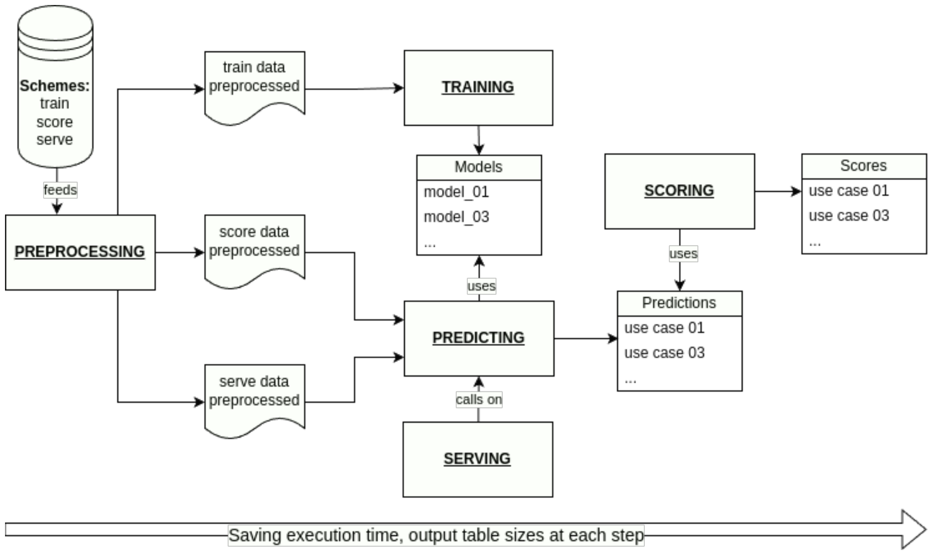

Fig. 1: Benchmark workflow is carried out by the five procedures preprocessing, training, predicting, scoring, and serving.

2. **Train**: Conducts model training using the preprocessed data and saves the model as a table in the database.
3. **Predict**: Uses the trained model and the preprocessed data from either score or serve schema to conduct inference.
4. **Score**: Evaluates the prediction results against true labels to ensure the model meets the minimum threshold; implementation via SQL.
5. **Serve**: Conducts serving of test data from serve schema by leveraging the Preprocess and Predict procedures.

All procedures are constructed using the procedural language PL/pgSQL, which offers powerful input and output control by utilizing arguments similar to those in function calls. Additionally, it supports loops and other typical functionalities found in procedural languages. We chose PL/pgSQL for its fine-grained control over the entire workflow, its reusability, and its potential for future extensibility. However, the core functionalities of the *preprocessing* and *scoring* steps in most use cases can be achieved with ANSI SQL.[2] An overview of the specific languages and extensions used at each step and for each use case is provided in Table 1. Note that PL/pgSQL is abreviated with PL, and instances where PL/pgSQL is used solely used to forward input arguments are considered as SQL.

[2] ANSI SQL conformity does not guarantee explicit portability, as the extent of the implementation of ANSI SQL syntax is platform-dependent.

Table 1: Minimum Requirement of Languages and Extensions at each step; serving is a combination of preprocessing and predicting, thus left out

Use Case	Preprocessing	Training	Predicting	Scoring
01	SQL	MADlib	MADlib	MADlib
03	PL, SQL	MADlib	MADlib, PL	SQL
04	SQL	SQL	SQL	SQL
06	SQL	MADlib, PL	MADlib, PL	SQL
07	SQL	MADlib, PL	MADlib, PL	SQL
08	MADlib, SQL	MADlib	MADlib	SQL
10	SQL	MADlib	MADlib	SQL

5.2 Implementation details for each Use Case

Use Case 01 - Clustering with K-Means++ The first use case represents a customer segmentation problem solved with the K-Means++ algorithm. The preprocess procedure uses common table expressions (CTEs) to aggregate the four tables *customer*, *order*, *lineitem*, and *order_returns*. It sums and filters out the relevant data for the training step. Following the TPCx-AI implementation, the training step utilizes two features, *ReturnRatio* and *Frequency*, to assign each vector to one of four clusters. For evaluation, we propose utilizing the silhouette function to evaluate the fitness of the clusters. It assesses how similar a vector is to its own cluster compared to other clusters by calculating the separation distance between the resulting clusters. The serve procedure then uses the pre-trained model to assign new vectors from a test set to the predefined clusters. Unlike the Python implementation of TPCx-AI, min-max scaling is now an integrated part of the processing procedure. It fits a custom implementation of a min-max function on the training dataset and uses the fitted function to transform the scoring and serving datasets into the fitted range.

Use Case 03 - Time Series Forecasting with ARIMA In use case 3, the tables *orders*, *product*, *lineitem*, and *store_department* are aggregated to compute the weekly sales figures. These aggregated sales data represent the training features. They are grouped by each store and each department within each store. The objective is to predict future weekly sales up to one year based on past two years' sales data. The Python implementation uses Holt-Winters' exponential smoothing, which is not available within the MADlib extension. Consequently, we opted for the ARIMA time series model, which is available in MADlib. Since the goal is to predict next year's sales figures for each store and department combination, a unique model is generated for each. Due to the large number of models required, we did not fine-tune the p, q, and d parameters for each individual model. This lack of parameter adjustment significantly degraded the models' performance, as demonstrated in the results section.

Use Case 04 - Binary Classification with Naive Bayes Use case 4 utilizes Naive Bayes classification for spam detection. As of May 2024, the Naive Bayes algorithm in the MADlib framework is in the early stages of development. Given the relatively low complexity of the algorithm, we opted for a custom implementation from scratch. Initially, our preprocessing step involves unnesting the *review* table of each text chunk into an array of stemmed word tokens. During the training step, utilizing Laplace Smoothing for zero probability adjustment, a lookup table for all tokens with their respective probabilities for the binary classes is created. In the prediction phase, probabilities for a test dataset are calculated using the previously built model. One of the significant challenges encountered during the implementation was the issue of numerical underflow, commonly faced in applications involving large-scale multiplication of small probabilities. Our training corpus consisted of approximately 250,000 total word tokens, with an average review containing about 100-150 words. The multiplication of these tiny probabilities representing each word resulted in the numeric underflow problem. By utilizing the logarithm of probabilities instead of the probabilities themselves, we effectively stabilized the computation.

Use Case 06 - Binary Failure Prediction with SVM Use Case 6 focuses on predictive maintenance, particularly predicting hard drive failures using SVM (Support Vector Machine) classification. In the preprocessing stage, cases are selected from the *failures* table where at least one failure has occurred. This selection is crucial for training the model to recognize patterns indicative of impending failures. Additionally, the preprocessing includes calculating the 'time to fail' (TTF) for each drive and setting the prediction label (True label) for the day before failure to 1, which is critical for predicting imminent hard drive failures. During the training phase, data is upsampled to address class imbalance, a common issue in datasets where failures are rare compared to non-failures. This balancing ensures that the SVM model is not biased toward the majority class. While the Python implementation of TPCx-AI leverages the `imblearn` package's ADASYN algorithm—which generates a weighted distribution of synthetic data with a stronger focus on harder-to-learn minority classes—we utilized MADlib's stratified sampling function for upsampling. This approach resulted in comparable performance, as there is only one minority class. For prediction, the test dataset from the *score* or *serve* schema is first standardized and then forwarded to MADlib's SVM classifier. For scoring, we implemented a custom metric function to calculate the Matthews Correlation Coefficient (MCC). Both use cases (Python and MADlib) utilize the Gaussian kernel for classification. To align with TPCx-AI's Python implementation, we changed the default lambda value to 1.0, as in the default implementation of Scikit-Learn. Here, $\lambda = \frac{1}{C}$ and C is set to 1.0. Furthermore, we increased the maximum number of iterations to 100,000 to match the unlimited (-1) setting in Scikit-Learn. We also increased the tolerance level from 1×10^{-10} to 1×10^{-3}. Lastly, the class weight parameter has been set to `"balanced"` [2].

Use Case 07 - Recommender System with SVD The objective of use case 7 is to perform product recommendation through collaborative filtering using matrix factorization. The training table *productrating* represents a highly sparse user-item rating matrix. We predict the unknown ratings by employing MADlib's low-rank matrix factorization (LMF) to fill in the blanks. During preprocessing, user IDs are incremented by one to ensure proper indexing. This step is critical as it prepares the data in the format required by the MADlib's factorization operations. The training procedure begins with calculating the dimensions of the user-item matrix before applying matrix factorization. Both MADlib's LMF and Python library `Surprise` use stochastic gradient descent (SGD) to optimize the latent factors in matrices $\boldsymbol{U}$ and $\boldsymbol{V}$. The aim is to minimize prediction errors on known ratings while generalizing to predict unknown ratings. We adjust the following parameters in line with the Python implementation: max rank r is increased from the default value 20 to 100, step size is decreased from 0.01 to 0.005, and number of iterations is increased from 10 to 20 [3]. The prediction procedure offers three options for filling the sparse user-item rating matrix. The first option utilizes MADlib's built-in `matrix_mult` function to multiply the dense matrices $\mathbf{U}$ and $\mathbf{V}$. While independent of the input matrices' sizes, it is computationally very slow. The second option leverages the `numpy` framework for matrix multiplication, which significantly improves computation speed but is constrained by memory size. The third and our preferred option uses MADlib's `array_dot` function for matrix dot product and computes only the required user-item combinations specified by the serve dataset.

Use Case 08 - Multi-class Classification with XGBoost Use Case 8's objective is to predict the outcome of the multi-label variable *trip_type* with XGBoost. The tables *order*, *lineitem*, *product*, and *store_dept* are consolidated during preprocessing. This procedure generates binary columns for each day of the week and for each department. In the final step of preprocessing, MADlib's `stratified_sampling` function is called for downsizing the preprocessed table if it exceeds 1GB.[3]. For training, we also adjusted the default training parameters to align with TPCx-AI's implementation. Notably, the training time was more than halved without significant loss in performance by using the histogram-based tree method. Unlike the `exact greedy algorithm`, histogram-based tree method significantly reduces the computational complexity and memory usage [7]. During inference, our test showed that XGBoost's predict function could not handle any input table exceeding 300MB in size. Therefore, downsizing via stratified sampling was applied at SF15.

Use Case 10 - Binary Classification with Logistic Regression Use Case 10 employs logistic regression for fraud detection. Compared to classification

[3] As of MADlib 2.1.0, XGBoost is still under early development. Its training function has an input limit of 1GB since all data is collected in a single segment and stored in one single cell[1]

with XGBoost, logistic regression is a much faster algorithm with good accuracy and high explainability. During preprocessing, the *financial_ account* table is joined with the *financial_ transactions*. The features used for training are the columns *business_ hour_ norm* and *amount_ norm*. These represent the normalized business hour and transaction amount, respectively. The Python implementation uses the LBFGS solver, a quasi-Newton method, particularly useful for large-scale optimization problems with limited memory [16]. The closest method available in the Apache MADlib logistic regression implementation is the conjugate gradient solver. While it doesn't use gradient information to approximate the inverse Hessian matrix as LBFGS does, it uses the conjugate directions of gradients for faster convergence [12]. The conjugate gradient solver achieved similar accuracy as the default optimizer IRLS and required considerable longer computation time. Thus, we opted for IRLS.

6 Experiments

An experiment is defined as the end-to-end execution of the seven implemented use cases at a specific scale factor. To minimize bias and ensure consistent performance, a system-wide restart is performed before initiating a new experiment, as PostgreSQL's performance may vary once the data has been pre-read. Runtime of use case 8's train procedure is not included in the analysis due to the input limit restriction as mentioned in the implementation section. A summary of all output tables for various stages is shown in Table 5.

6.1 Data Loading - CSV to PostgreSQL

Before initiating any experiments, data generated using PDGF at different scale factors needs to be loaded into the PostgreSQL database. We utilize the `Psycopg` library to copy the CSV files into predefined empty tables, which are then indexed to enhance performance. This step closely resembles the loading test (T_{LD}) of TPCx-AI. A fitted linear regression with an R^2 score of 0.98 indicates a linear relationship between dataset size and loading time.

6.2 Performance Comparison against the Python Implementation

The Power Training Test of TPCx-AI involves the following steps: reading the respective CSV and PSV files, preprocessing the loaded data with Python libraries such as `Pandas` and `Numpy`, and model training. To facilitate a fair comparison, we combine the runtime results from our *Preprocess* and *Train* procedures and refer to it as *train* henceforth.

The Power Serve Test of TPCx-AI comprises data loading, preprocessing, and inferencing with the trained model. TPCx-AI evaluates serving performance in a single-stream scenario and in a multi-stream version, known as the Power Serving Test and the Serving Throughput Test, respectively. Our *Serve* procedure comprises *Preprocess* and *Predict* procedures, making them comparable. As we

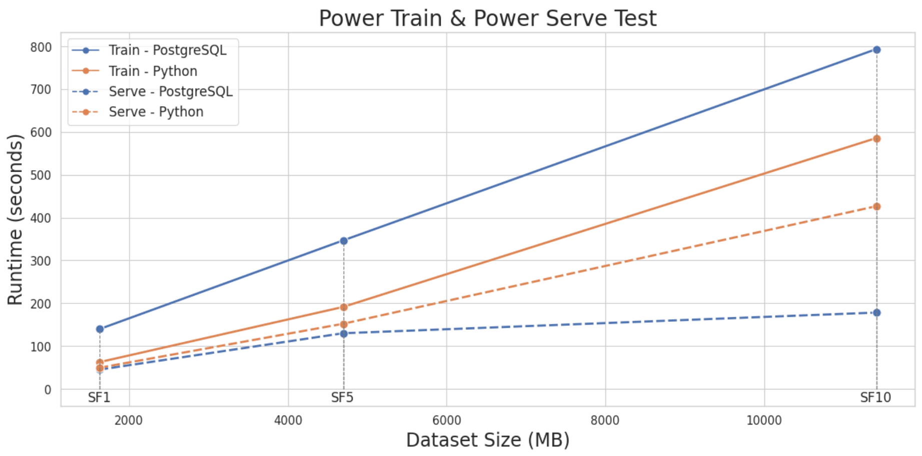

Fig. 2: Comparison of training and serving in TPCx-AI's Python benchmark and PostgreSQL with Apache MADlib

Table 2: Performance comparison between PostgreSQL and Python in seconds at different scaling factors.

Use Case	**Step**	**SF1 PostgreSQL**	**SF1 Python**	**SF5 PostgreSQL**	**SF5 Python**	**SF10 PostgreSQL**	**SF10 Python**
uc01	train	26.151	14.184	33.644	42.676	153.903	112.377
uc03	train	36.318	17.306	61.718	44.630	120.630	111.159
uc04	train	15.670	11.350	52.177	34.455	96.853	58.895
uc06	train	2.096	0.779	5.541	21.442	11.801	183.954
uc07	train	8.240	1.200	26.673	4.364	47.484	7.485
uc10	train	51.319	17.829	166.987	43.860	362.780	111.882
$\sum$train		139.794	62.648	346.74	191.427	793.451	585.752
geometric mean train		15.371	5.995	37.255	25.412	84.558	69.565
uc01	serve	1.268	1.509	3.901	3.785	9.424	9.915
uc03	serve	4.638	2.560	6.449	2.986	8.525	3.970
uc04	serve	2.203	2.131	4.863	5.022	8.473	7.765
uc06	serve	0.113	0.518	0.790	11.631	1.899	81.296
uc07	serve	0.036	0.597	0.171	1.673	0.288	2.891
uc10	serve	1.516	1.794	6.672	4.679	15.689	11.460
$\sum$serve		9.774	9.109	22.846	29.776	44.298	117.297
geometric mean serve		0.656	1.288	2.190	4.158	4.244	9.681
$\sum$total		149.568	71.757	369.586	221.203	837.749	703.049
total geometric mean		3.176	2.779	9.032	10.279	18.943	25.951

have conducted our tests using a single stream, we compare our serving test results with the results of TPCx-AI's single-stream benchmark.

Table 2 displays a detailed comparison between both for each use case up to scale factor 10. The Python implementation outperforms PostgreSQL with Apache MADlilb on an aggregated level. The primary discrepancy arises in use case 10: `Scikit-Learn`'s logistic regression with the LBFGS optimizer performs significantly faster than MADlib's implementation of logistic regression with IRLS. Figure 3 visually represents this information using a radar chart. Although model training speed is generally faster in Python, PostgreSQL's strength lies in model serving, particularly at higher scale factors. This advantage is clearly illustrated in Figure 2.

Table 3: Runtime comparison of loading and preprocessing training data in seconds between PostgreSQL and Python at SF15. *failed preprocessing

Use Case	Step	SF15 PostgreSQL	SF15 Python
uc01	preprocess (train)	142.278	199.077
uc03	preprocess (train)	95.437	158.588
uc04	preprocess (train)	84.693	4.642
uc06	preprocess (train)	2.161	5.892
uc07	preprocess (train)	0.022	0.881
uc08	preprocess (train)	634.553	NaN*
uc10	preprocess (train)	97.334	159.816

6.3 Qualitative Metrics

With the exception of use case 3, almost all use cases on different scale factors achieve the minimum threshold requirement.[4] Just as in the Python implementation of TPCx-AI, metrics for most use cases remain constant, with few metrics improving with increasing data size, as shown in Table 5.

7 Conclusion and Future Work

In this paper, we demonstrated the machine learning capabilities of PostgreSQL with the Apache MADlib extension. PostgreSQL exhibited outstanding performance in data aggregation and manipulation, while MADlib proved effective for classical machine learning algorithms, particularly for inference tasks. However, in-database ML with MADlib has limitations, especially in deep learning. Despite support for TensorFlow, there are limited fine-tuning possibilities, such as training for regression problems.

[4] As described in the implementation section, the poor performance of use case 3 is due to the usage of ARIMA instead of exponential smoothing without adjusting p,d,q values for each individual model.

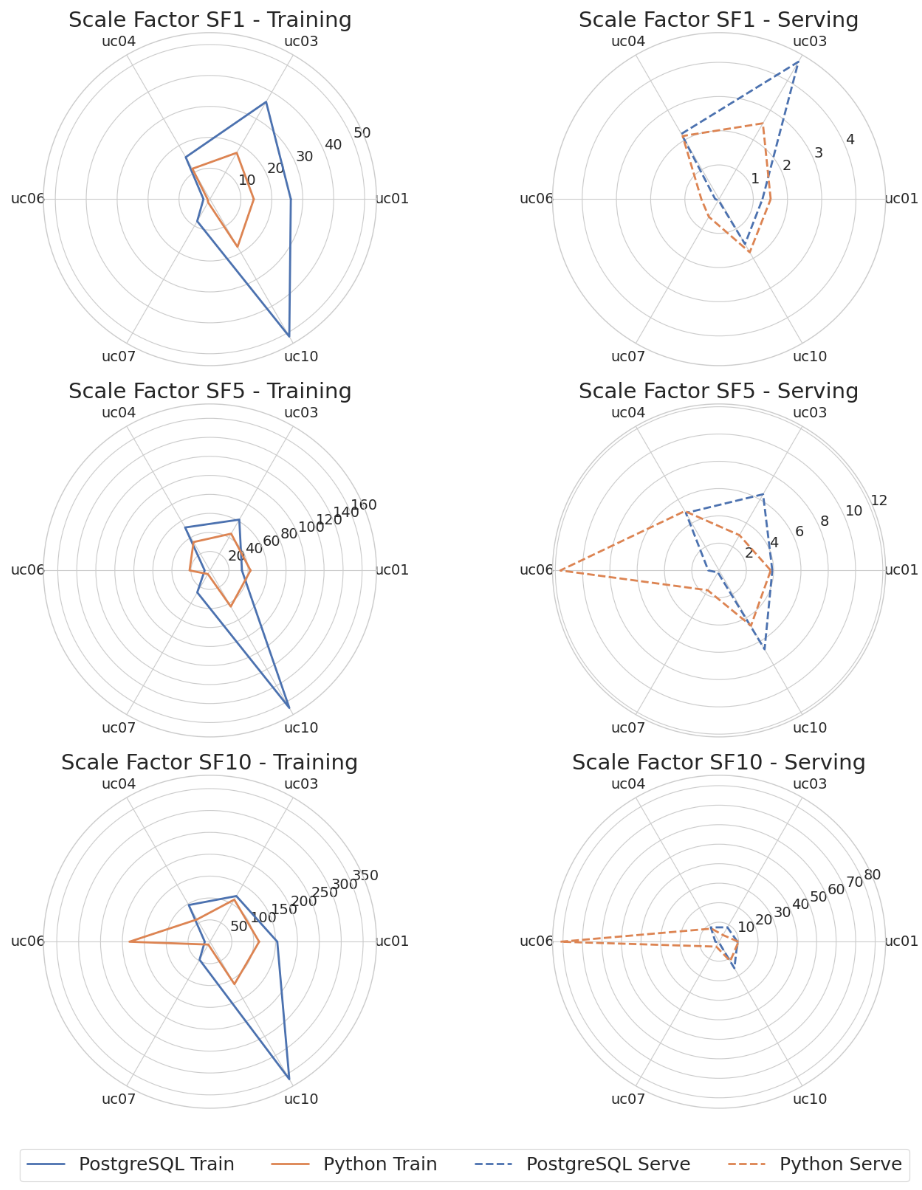

Fig. 3: Detailed comparison of TPCx-AI's implementation of each use case vs PostgreSQL with Apache MADlib. Runtime in seconds. Use case 08 excluded.

Table 4: Output tables from various procedures for all use cases at different SF. Scoring results have mostly identical sizes due to the fixed size of true label data. Output for uc08_preprocessed stays constant due to down sampling after data aggregation.

Table name	**SF1 size**	**SF5 size**	**SF10 size**	**SF15 size**
uc03_score_results	1896 kB	1960 kB	1976 kB	2008 kB
uc04_score_results	1184 kB	1184 kB	1184 kB	1184 kB
uc06_score_results	424 kB	424 kB	424 kB	424 kB
uc07_score_results	16 kB	16 kB	16 kB	16 kB
uc08_score_results	21 MB	21 MB	21 MB	21 MB
uc10_score_results	44 MB	44 MB	44 MB	44 MB
uc01_serve_results	544 kB	1632 kB	2696 kB	3704 kB
uc03_serve_results	1608 kB	1792 kB	2128 kB	2288 kB
uc04_serve_results	792 kB	2752 kB	4856 kB	6680 kB
uc06_serve_results	216 kB	1944 kB	5544 kB	10 MB
uc07_serve_results	976 kB	3896 kB	6880 kB	9392 kB
uc08_serve_results	131 MB	377 MB	377 MB	377 MB
uc10_serve_results	31 MB	93 MB	236 MB	324 MB
uc01_train_preprocessed	11 MB	33 MB	60 MB	82 MB
uc03_train_preprocessed	3408 kB	3864 kB	6848 kB	7376 kB
uc04_train_preprocessed	133 MB	460 MB	813 MB	1114 MB
uc06_train_preprocessed	1288 kB	11 MB	30 MB	57 MB
uc07_train_preprocessed	7408 kB	29 MB	50 MB	69 MB
uc08_train_preprocessed	858 MB	858 MB	858 MB	858 MB
uc10_train_preprocessed	479 MB	1436 MB	3643 MB	4987 MB
uc01_model	16 kB	16 kB	16 kB	16 kB
uc03_model	8096 kB	9024 kB	11 MB	11 MB
uc04_model	960 kB	960 kB	960 kB	992 kB
uc06_model	16 kB	16 kB	16 kB	16 kB
uc07_model	11 MB	34 MB	57 MB	78 MB
uc08_model	2288 kB	2240 kB	2240 kB	2200 kB
uc10_model	16 kB	16 kB	16 kB	16 kB

Table 5: Scoring metrics for the 7 use cases at different scale factors

Use Case	SF1 Score	SF5 Score	SF10 Score	SF15 Score	Threshold
1	0.652160	0.642697	0.605474	0.618169	None
3	70.543810	72.340388	77.657227	79.681503	< 5.40
4	0.695846	0.702113	0.705238	0.694204	> 0.65
6	0.362314	0.438165	0.463538	0.468870	> 0.19
7	1.570248	1.443262	1.498282	1.500000	< 1.80
8	0.641002	0.667867	0.652621	0.654225	> 0.65
10	0.816386	0.817302	0.816795	0.815871	> 0.70

Additionally, MADlib lacks implementations of some popular machine learning algorithms like Holt-Winters exponential smoothing and XGBoost, the latter of which is still in early development. The training time for many algorithms is slower compared to their Python counterparts. However, MADlib's SVM (Support Vector Machine) significantly outperforms scikit-learn's implementation, indicating the potential for speed improvements in Apache MADlib's algorithms.

One key advantages of in-database machine learning is the minimized data movement. During preprocessing, up to SF15, PostgreSQL scales linearly and outperforms Python's loading and preprocessing (as shown in Table 3), highlighting the efficiency of in-database ML for data handling and preparation.

The TPCx-AI benchmark has proven to be a valuable tool for examining the ML capabilities of database systems. For future work on Apache MADlib, we suggest focusing on performance optimization for selected algorithms, expanding algorithm coverage, and enhancing deep learning support. Additionally, testing other types of database systems with TPCx-AI's benchmark use cases would further expand the applicability and understanding of in-database machine learning.

In summary, PostgreSQL with Apache MADlib shows strong potential for in-database ML despite the mentioned limitations. At SF5 and beyond, the total geometric mean for training plus serving is smaller than that of its Python counterpart. Its advantages in inference performance, data handling efficiency, and integration with existing database infrastructure make it a promising solution for organizations looking to leverage ML capabilities within their databases.

References

1. MADlib: XGBoost — madlib.apache.org. `https://madlib.apache.org/docs/latest/group__grp__xgboost.html`, [Accessed 20-05-2024]
2. Support Vector Machines — scikit-learn.org. `https://scikit-learn.org/1.2/modules/svm.html`, [Accessed 20-05-2024]
3. Welcome to Surprise documentation! Surprise 1 documentation — surprise.readthedocs.io. `https://surprise.readthedocs.io/en/stable/index.html`, [Accessed 20-05-2024]

4. Bayer, R., Tøttrup, J.V., Tözün, P.: Tpcx-ai on nvidia jetsons. In: Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers. p. 49–66. Springer-Verlag, Berlin, Heidelberg (2023), https://doi.org/10.1007/978-3-031-29576-8_4
5. Boehm, M., Kumar, A., Yang, J.: Data Management in Machine Learning Systems. Morgan & Claypool Publishers (2019)
6. Brücke, C., Härtling, P., Palacios, R.D.E., Patel, H., Rabl, T.: Tpcx-ai - an industry standard benchmark for artificial intelligence and machine learning systems. Proc. VLDB Endow. **16**(12), 3649–3661 (aug 2023), https://doi.org/10.14778/3611540.3611554
7. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2016), https://api.semanticscholar.org/CorpusID:4650265
8. Cohen, J., Dolan, B., Dunlap, M., Hellerstein, J.M., Welton, C.: Mad skills: new analysis practices for big data. Proc. VLDB Endow. **2**(2), 1481–1492 (aug 2009). https://doi.org/10.14778/1687553.1687576, https://doi.org/10.14778/1687553.1687576
9. Erdelt, P.K.: A framework for supporting repetition and evaluation in the process of cloud-based dbms performance benchmarking. In: Nambiar, R., Poess, M. (eds.) Performance Evaluation and Benchmarking. pp. 75–92. Springer International Publishing, Cham (2021)
10. Erdelt, P.K., Jestel, J.: Dbms-benchmarker: Benchmark and evaluate dbms in python. Journal of Open Source Software **7**(79), 4628 (2022), https://doi.org/10.21105/joss.04628
11. Hellerstein, J.M., Ré, C., Schoppmann, F., Wang, D.Z., Fratkin, E., Gorajek, A., Ng, K.S., Welton, C., Feng, X., Li, K., Kumar, A.: The madlib analytics library: or mad skills, the sql. Proc. VLDB Endow. **5**(12), 1700–1711 (aug 2012), https://doi.org/10.14778/2367502.2367510
12. Hestenes, M.R., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of research of the National Bureau of Standards **49**, 409–435 (1952), https://api.semanticscholar.org/CorpusID:2207234
13. Kim, H.S., Kim, D., Seo, B., Hwang, S.: Tpcx-ai: First adopter's experience report. In: Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers. p. 120–126. Springer-Verlag, Berlin, Heidelberg (2023), https://doi.org/10.1007/978-3-031-29576-8_9
14. Kumar, A., Boehm, M., Yang, J.: Data management in machine learning: Challenges, techniques, and systems. In: Proceedings of the 2017 ACM International Conference on Management of Data. p. 1717–1722. SIGMOD '17, Association for Computing Machinery, New York, NY, USA (2017), https://doi.org/10.1145/3035918.3054775
15. Liu Olesiuk, Y., Hodak, M., Ellison, D., Dholakia, A.: More the merrier: Comparative evaluation of tpcx-ai and mlperf benchmarks for ai. In: Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers. p. 67–77. Springer-Verlag, Berlin, Heidelberg (2023), https://doi.org/10.1007/978-3-031-29576-8_5
16. Nocedal, J.: Updating quasi-newton matrices with limited storage. Mathematics of Computation **35**, 773–782 (1980), https://api.semanticscholar.org/CorpusID:9033333

17. Patel, H., Ufa, K., Nah, S., Raina, A., Escobar, R.: Preliminary scaling characterization of tpcx-ai. In: Performance Evaluation and Benchmarking: 14th TPC Technology Conference, TPCTC 2022, Sydney, NSW, Australia, September 5, 2022, Revised Selected Papers. p. 78–93. Springer-Verlag, Berlin, Heidelberg (2023), https://doi.org/10.1007/978-3-031-29576-8_6
18. Watson, A., Babu, D.S.V., Ray, S.: Sanzu: A data science benchmark. In: 2017 IEEE International Conference on Big Data (Big Data). pp. 263–272 (2017). https://doi.org/10.1109/BigData.2017.8257934
19. Zhang, Y., McQuillan, F., Jayaram, N., Kak, N., Khanna, E., Kislal, O., Valdano, D., Kumar, A.: Distributed deep learning on data systems: a comparative analysis of approaches. Proceedings of the VLDB Endowment **14**(10). https://doi.org/10.14778/3467861.3467867, https://par.nsf.gov/biblio/10337018