TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems

Christoph Brücke
bankmark
Germany
christoph.bruecke@bankmark.de

Philipp Härtling
bankmark
Germany
philipp.haertling@bankmark.de

Rodrigo D Escobar Palacios
Intel
Hillsboro, Oregon
rodrigo.d.escobar.palacios@intel.com

Hamesh Patel
Intel
Hillsboro, Oregon
hamesh.s.patel@intel.com

Tilmann Rabl
Hasso Plattner Institute, University of Potsdam, bankmark
Germany
tilmann.rabl@hpi.de

ABSTRACT

Artificial intelligence (AI) and machine learning (ML) techniques have existed for years, but new hardware trends and advances in model training and inference have radically improved their performance. With an ever increasing amount of algorithms, systems, and hardware solutions, it is challenging to identify good deployments even for experts. Researchers and industry experts have observed this challenge and have created several benchmark suites for AI and ML applications and systems. While they are helpful in comparing several aspects of AI applications, none of the existing benchmarks measures end-to-end performance of ML deployments. Many have been rigorously developed in collaboration between academia and industry, but no existing benchmark is standardized.

In this paper, we introduce the TPC Express Benchmark for Artificial Intelligence (TPCx-AI), the first industry standard benchmark for end-to-end machine learning deployments. TPCx-AI is the first AI benchmark that represents the pipelines typically found in common ML and AI workloads. TPCx-AI provides a full software kit, which includes data generator, driver, and two full workload implementations, one based on Python libraries and one based on Apache Spark. We describe the complete benchmark and show benchmark results for various scale factors. TPCx-AI’s core contributions are a novel unified data set covering structured and unstructured data; a fully scalable data generator that can generate realistic data from GB up to PB scale; and a diverse and representative workload using different data types and algorithms, covering a wide range of aspects of real ML workloads such as data integration, data processing, training, and inference.

1 INTRODUCTION

Machine learning (ML) has been shown to be beneficial in industry and academia. Tasks that were considered very complex to solve with software systems, such as image or voice recognition, can be solved with high accuracy with ML-based artificial intelligence (AI) solutions [34]. While many large companies have shown surprising results in various applications, this is hard to replicate for companies without well-funded and well-staffed AI departments. Today, improvements in accuracy and architecture come at costs that only major players can afford and that are not feasible or even reproducible for small scale deployments [11].

Contributing to this problem, current AI solutions are highly diverse and due to the great interest in the field, many new systems with different strategies and goals are developed. This creates a need for standardized methods to compare systems and solutions. In the past, benchmarks have been pivotal for both, performance improvements in common applications and unification of interfaces for these applications. Previous database benchmarks by the Transaction Processing Performance Council (TPC) are a good example of this, which is shown by the many publications in database research that use these benchmarks as validation. Although they are a simplification of real world applications, the benchmarks serve as a common abstraction for use case scenarios and make systems and research comparable.

Researchers and practitioners have started several efforts to build similar benchmarks for AI domains. Most of these focus on the core ML training aspect, either from a micro-benchmark perspective, such as DeepBench [22] or evaluating the training of different ML models, e.g., MLBench [17]. While model training is at the heart of many AI applications, it does not reflect the various challenges AI applications face. More holistic approaches also incorporate serving tasks, e.g., MLPerf [18], which allows for the evaluation of training or serving for a variety of ML models, or preprocessing and postprocessing stages, such as debench [7], which evaluates specific preprocessing tasks, such as data pruning.

1Serving is also commonly known as Inference in the AI terminology.
In this paper, we go beyond these approaches and introduce the TPC Express Benchmark for Artificial Intelligence (TPCx-AI), a benchmark for end-to-end AI systems. TPCx-AI is based on ADABench [29] and has recently been standardized by the TPC [4]. TPCx-AI’s workload comprises 10 use cases, implemented as end-to-end AI processing pipelines that, unlike other AI benchmarks, include data ingestion, preprocessing, training, serving, and post-processing stages. Figure 1 shows the set of tasks involved in a typical end-to-end AI processing pipeline. TPCx-AI’s use cases are based on diverse retail scenario tasks with various data types and ML models, both traditional and deep learning. We use a study by McKinsey Analytics [1] on potential use cases for AI and ML techniques in retail and other verticals as an inspiration for the workloads.

TPCx-AI is released with a full implementation, also known as the toolkit or simply the kit. This kit contains a synthetic data generator, based on the Parallel Data Generation Framework (PDGF) [30], a driver that manages the complete execution of the benchmark, as well as two reference implementations, one based on Apache Spark [38] and one based on established Python libraries. Other implementations can be added as well upon review and approval by the TPC. Despite being a young standard benchmark, there already exist 14 official benchmark submissions to the TPC, as of June 2023.

In this paper, we give an overview of TPCx-AI, the first industry standard benchmark for end-to-end AI and ML systems and make the following contributions:

- We give a detailed introduction to TPCx-AI. We describe its workload, data sets, run rules, and metrics.
- We describe the internal benchmark implementation and compare the Spark- and Python-based versions.
- We analyze the benchmark data set and workload both qualitatively and experimentally.

The rest of this paper is structured as follows. Next, we discuss related work. Section 3 gives an overview of the benchmark and describes the workload, data model, run rules, and metrics. We analyze the data set in Section 4. Section 5 presents the benchmark kit and the two workload implementations. In Section 6, we execute the benchmark and discuss the results of both implementations. We present future extensions of the benchmark in Section 7 and conclude with future work in Section 8.

2 RELATED WORK

In this section, we will give an overview of related work. For a recent survey on several machine learning, big data, and high performance computing benchmarks see also Ihde et al. [15].

Due to the high interest in ML, many benchmarks have been developed to evaluate ML systems and applications. These can be categorized in benchmark data sets, micro-benchmarks, benchmark suites, component benchmarks, and application-level benchmarks.

Benchmark data sets have a long history in ML research. Well known examples are ImageNet [5] and WordNet [20], as well as various collections of ML data sets, such as the UCI Machine Learning Repository2, OpenML [36], and the Penn Machine Learning Benchmarks [31]. These data sets are typically used to evaluate the accuracy of ML algorithms and approaches rather than their performance. While this has helped tremendously to improve ML algorithms, it does not provide the full picture required in practical deployments, where a trade-off in accuracy and latency is always required. Several benchmark proposals are based on these data sets, e.g., MLBench [17] and MLPerf [18].

An example of a micro-benchmark is DeepBench [21, 22], which benchmarks basic deep learning operations, such as matrix multiplication, all-reduce, or convolutions. Another example is SLAB [35], a benchmark for linear algebra primitives. Like other micro-benchmarks, DeepBench and SLAB can give deep insights into performance behaviour of the underlying hardware for typical ML operations, but the results are hard to interpret for end-to-end scenarios in real world deployments.

Benchmark suites are collections of kernels, applications, tools, or other workload units, designed to represent certain workload scenarios in application domains. An example of a kernel based benchmark suite is High Performance Linpack3 (HPL), a collection of kernels for numerical calculations to measure raw compute performance. Like many benchmark suites, HPL extensions have added ML workloads (e.g., HPL-AI4) to their portfolio. Another example is BigDataBench [8], which includes AI workloads in its current

---

3High Performance Linpack - https://www.netlib.org/benchmark/hpl/
4HPL-AI - https://cil.biboss.io/hpl-ai
version\textsuperscript{5}. Some benchmark suites contain end-to-end application workloads, but are built to measure hardware performance rather than complete deployments.

Most research on AI has focused on ML training, which is also true for the majority of ML benchmarks. From a benchmarking perspective, this can be seen as one component (among several others) of the complete ML pipeline. Researchers have identified other tasks of ML pipelines to be relevant for end-to-end performance of ML deployments and systems \cite{28, 37}, and some benchmarks have been developed especially for those other tasks. Example of tasks are data cleaning, such as presented by CleanML \cite{16}, a benchmark for data cleaning and ML; AutoML Benchmark \cite{10}, which benchmarks automated ML. In addition, dbench \cite{7} is a benchmark that specifically focuses on ML pipeline stages other than training. Currently, the benchmark defines three components for data cleaning, pruning, and slicing.

Closest to TPCx-AI are MLPerf \cite{18} and its predecessor DAWN-Bench \cite{3}, which both include training and serving tasks and proposed novel metrics for time to accuracy and time per serving. Both metrics are relevant for real world deployments and are included in the TPCx-AI metric. Unlike TPCx-AI, both benchmarks do not include data ingestion, preprocessing or postprocessing stages, and users can choose to evaluate only training or serving, or both. To the best of our knowledge, TPCx-AI is the only official industry standard benchmark, which covers end-to-end ML and AI workloads at an application level.

3 BENCHMARK OVERVIEW

In this section, we give an overview of the benchmark specification, which includes the workload (use case pipelines and data set), as well as the run rules and benchmark metrics.

Major design goals for the benchmark were that it is realistic, representative, and relevant for ML and AI deployments. Section 3.1 gives a brief introduction to TPC benchmarks in general. In Section 3.2 we describe the TPCx-AI data model. Section 3.3 gives a detailed description of the ten use cases and their pipelines. Section 3.4 explains run rules and the different measured and unmeasured tests of the benchmark. Finally, Section 3.5 and 3.6 define TPCx-AI’s quality and performance metric AIU/Cpm@$SF$.

3.1 TPC Benchmarks

The Transaction Processing Performance Council (TPC) is a consortium with a history of more than 30 years developing successful benchmarks for data processing. Currently there are 10 active TPC benchmarks for different areas of data processing and data analysis. For instance, classical online analytical processing is benchmarked in TPC-H and TPC-DS \cite{23, 24}, data integration systems can be evaluated using TPC-DI \cite{26}, and TPCx-BB is a big data analytics benchmark \cite{9}, which also includes several machine learning tasks. TPC benchmarks are typically used by member companies (e.g. original equipment manufacturers or independent software vendors) to demonstrate and promote the features and capabilities of their hardware or software systems when processing certain types of workloads. Companies that submit benchmark results for official publication are also known as test sponsors.

Each TPC benchmark belongs to one of two categories: Express or Enterprise. Both Express and Enterprise benchmarks include a specification document when released. The specification document contains information about the benchmark design, run rules, as well as implementation details, among others. In addition to the specification document, Express benchmarks include a kit released by the TPC along with the benchmark’s specification document \cite{14}. Enterprise benchmarks, on the other hand, need to be implemented by the sponsor based on the specification document. The TPC introduced Express benchmarks in 2013 in an effort to make their benchmarks easier to adopt.

The following aspects are also integral to TPC benchmarks:

Audit requirement Before TPC benchmark results can be considered official they need be reviewed by an independent TPC certified auditor that will inspect the results thoroughly to confirm that all benchmark rules were followed and thus obtained under fair conditions to be compared to previous and future results.

System Under Test pricing In addition to reporting the performance score obtained after running the benchmark, users also need to report in detail the price of the system under test (SUT) and its components (software and hardware) used to obtain their performance score \cite{13}. TPC benchmarks usually consolidate the price of the SUT under their price/performance metric that needs to be included in the results. This feature of TPC benchmarks allows for consumers to get a clearer perspective on the total cost of the system required to achieve a certain performance score. Recently, special clauses have been added to the pricing specification to permit running TPC benchmarks in the cloud rather than only in on-site deployments.

3.2 Data Model

The core of the TPCx-AI data model is relational, i.e., there are multiple tables with relationships to each other. Figure 2 illustrates the different schema parts and highlights the relationships between the tables as well as their usage by the individual use cases. The relational schema represents an analytical database with orders consisting of line items as common in data warehouses. Additionally, several other relational tables give more insight into customers, their financial transactions, products, and marketplace prices. Besides the structured data, there are multiple unstructured data sets in the model. This also includes text, as in product reviews, as well as, audio files and customer images. Each use case utilizes a subset of the data model as shown in Figure 2. All tables are generated scalably using the synthetic data generator PDGF \cite{30}, which we extended to enable image and audio generation.

The complete data set is scaled according to a scale factor ($SF$). Users specify the $SF$ prior to running the benchmark to indicate the dataset size in GB for which they want to run the workload. Scale factors that can be used for official publications range from 1 GB up to several Petabytes in steps of 3 and 10 (i.e., 1, 3, 10, ...). The relational tables include common SQL data types, as well as text, and each table has between 2 and 12 columns. The audio files are WAV files and the images PNG files. Using synthetically generated

Table 1: Use Case Overview

<table>
<thead>
<tr>
<th>ID</th>
<th>use case</th>
<th>Type</th>
<th>Class</th>
<th>Data</th>
<th>Algorithm</th>
<th>DL</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC01</td>
<td>Customer Segmentation</td>
<td>Triaging</td>
<td>Clustering</td>
<td>Number</td>
<td>k-Means</td>
<td></td>
</tr>
<tr>
<td>UC02</td>
<td>Call Transcription</td>
<td>Convert Unstr. Data</td>
<td>Classification</td>
<td>Audio</td>
<td>Convolutional Neural Network</td>
<td>X</td>
</tr>
<tr>
<td>UC03</td>
<td>Sales Forecasting</td>
<td>Forecasting</td>
<td>Regression</td>
<td>Number</td>
<td>Holt-Winters</td>
<td></td>
</tr>
<tr>
<td>UC04</td>
<td>Spam Detection</td>
<td>Discover Anomalies</td>
<td>Classification</td>
<td>Text</td>
<td>Naïve Bayes</td>
<td></td>
</tr>
<tr>
<td>UC05</td>
<td>Price Prediction</td>
<td>Predictive Analytics</td>
<td>Regression</td>
<td>Text</td>
<td>Recurrent Neural Network</td>
<td>X</td>
</tr>
<tr>
<td>UC06</td>
<td>Hardware Failure</td>
<td>Predictive Maintenance</td>
<td>Classification</td>
<td>Number</td>
<td>Support Vector Machines</td>
<td></td>
</tr>
<tr>
<td>UC07</td>
<td>Product Rating</td>
<td>Hyper-Personization</td>
<td>Classification</td>
<td>Number</td>
<td>Collaborative Filtering</td>
<td></td>
</tr>
<tr>
<td>UC08</td>
<td>Classification of Trips</td>
<td>Triaging</td>
<td>Classification</td>
<td>Image</td>
<td>CNN &amp; Logistic Regression</td>
<td>X</td>
</tr>
<tr>
<td>UC09</td>
<td>Facial Recognition</td>
<td>Hyper-Personization</td>
<td>Classification</td>
<td>Number</td>
<td>Logistic Regression</td>
<td></td>
</tr>
<tr>
<td>UC10</td>
<td>Fraud Detection</td>
<td>Discover Anomalies</td>
<td>Classification</td>
<td>Number</td>
<td>Logistic Regression</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: TPCx-AI Data Schema

Data also mitigates the risks of running into ethical or privacy issues, and enables the generation of correct ground truth data. The data files are customer conversations, which are initially generated as text files and then converted to audio using a text-to-speech conversion tool. The same is true for user images, which are generated in a deterministic manner based on the user ID in different conditions using a face generator.

For the training, serving, and scoring tests of the benchmark, separate data sets are generated. This is done by facilitating the descriptive nature of a PDGF data generator. For all three data sets the same data description is used but with a different seed and scaling factor. Changing the seed will give a different data set but with the same structure and data distributions. We generate three separate data sets in order to prevent information leaks and minimize attack surface for unfair benchmark users. The training data set is used in the Power Training Test and contains all tables with all columns and hence all features necessary for training as well as the labels. The training data set grows proportionally to the scaling factor. The serving data set is to be used for serving and consist of all the tables and columns but without the labels. Because this is different from the training data set, simply memorizing the training data will not help during serving. The serving data set is used for the Power Serving Test as well as the Serving Throughput Test. The last data set is the scoring data set and is conceptually a combination of the latter two data sets. As a first step a data set containing only the features but none of the labels is generated and must be used during the Scoring Test but in the same environment as the Power Serving Test. Afterwards the labels are generated in the benchmark driver to calculate the quality metric.

The data distributions within the data sets are modeled after real data sets or carefully designed to have certain characteristics. Some attributes are highly skewed whereas others contain a lot of noise. A good example for a skewed data set is the failures event log. This contains log event for different hard drives, one per day, monitoring the health status of hard drive using S.M.A.R.T. data. For the training case the failures log also contains a flag indicating if a hard drive failed. Under real conditions a hard drive rarely fails and if so it mostly fails catastrophically, which means there is a single event out of thousands which is a failure. Examples for noisy data sets are the financial_transactions and product_rating relations. These contain fraudulent transactions and spam reviews. But they also contain transactions and reviews that share the same characteristics as the fraudulent or spammy ones but are in fact valid transactions and reviews. Adding such noise prevents the models from characterizing fraud or spam too aggressively.

3.3 Workload

TPCx-AI’s workload consists of ten use cases, each covering a complete end-to-end ML pipeline as outlined by Polyzotis et al. [27], dealing with the technical challenges in ML systems [32]. An abstract ML pipeline can be seen in Figure 1. Unlike previous benchmarks, TPCx-AI's workloads cover all steps in this pipeline, although not all steps are covered in all pipelines. The use cases were inspired by real world use cases and based on an analysis by McKinsey Analytics [1]. The workload covers traditional techniques as well as deep learning models. Each use case solves a specific ML challenge. In the following, we give an overview of the ten use cases, which are also summarized in Table 1.
**UC01 - Customer Segmentation.** In the first use case, customers are segmented based on their shopping behavior. It uses k-means clustering on the customer, order, lineitem, and order_returns tables to identify different types of customers. In the preprocessing step, duplicates and null values are removed, then in the training, the customers are clustered unsupervised by k-means. The clusters are used while serving to assign new users to groups.

**UC02 - Customer Conversation Transcription.** In UC02, transcripts of user calls are generated using audio to text conversion. This is done using the Deep Speech recurrent neural network (RNN) model [12]. In the preprocessing step, the audio data is loaded and resampled to a common sample rate (16kHz). Then Mel Frequency Cepstral Coefficients (MFCC) are calculated from the raw audio and used to train the deep neural network infrastructure. During serving the same preprocessing steps are applied and the already trained model is used to generate transcriptions. The transcripts are then used to compute word error rates comparing to the ground truth data in the scoring test.

**UC03 - Sales Forecasting.** The objective in UC03 is to forecast the weekly sales for each department in each store given a limited history of sales data for the second to last year in the data set. The pipeline uses the tables orders, product, lineitem, and store_department. The use case utilizes an exponential smoothing technique (Holt-Winters) for time series forecasting. In the acquisition stage, the tables are loaded and joined, then aggregates for weekly sales are generated in preprocessing. The training fits the Holt-Winters model, which is used to forecast sales in the serving stage for up to one year per department per store. The scoring test checks the accuracy of the results.

**UC04 - Spam Detection.** In UC04, product reviews are searched for spam entries. This use case uses the product_rating text data. Our specification uses a classical Naïve Bayes model to identify spam entries. In the acquisition step, reviews are loaded and in the cleaning step, duplicates are eliminated. The text is transformed to n-grams and vectors in preprocessing and the model is fit during training. While serving, new reviews are classified as ham or spam and the resulting predictions are scored using Matthews correlation coefficient.

**UC05 - Price Prediction.** UC05 performs price predictions for individual retail items using a Recurrent Neural Network on the marketplace table. The prediction is based on the brand, product name, and description on an online market place. Initially, duplicates and null values are eliminated and the model is trained on the data. During serving, item prices are predicted and the predictions are scored using Root Mean Squared Log Error.

**UC06 - Hardware Failure.** In UC06 hardware failures of disk drives are predicted based on Self-Monitoring, Analysis, and Reporting Technology (SMART) data in the failures log data set. While this is not a pure retail use-case it is a typical infrastructure task and relevant in any larger hardware setup. The initial preprocessing performs duplicate and null value removal. A support vector machine is trained based on known data and failures. In the serving stage, impending failures are predicted and scored using the F-score metric.

**UC07 - Product Rating.** In UC07, cross-selling is improved by giving "next-to-buy" recommendations. Based on previously bought products, we give recommendations for products that the customer might also be interested in. The recommendation are found in the product ratings by comparing customers (by their products) and/or products (by their customers). The ratings are loaded and transformed into a numerical format without duplicates. In the training step, matrix factorization is used and an alternating least squares regression between the item and user matrix. Predictions for user item pairs are served and scored using mean absolute error.

**UC08 - Classification of Trips.** In a brick and mortar store setup, it is useful to classify shopping trips into trip types. Examples of trip types are: a customer may make a small daily dinner trip, a weekly large grocery trip, a trip to buy gifts for an upcoming holiday, or a seasonal trip to buy clothes. In data acquisition, the shopping history is generated by joining the tables order, lineitem, and product. Items are aggregated and categorical values in the data are binarized (after data cleaning). We train gradient boosted trees to predict the trip type in the serving stage based on active shopping sessions. The scoring computes the classification accuracy.

**UC09 - Facial Recognition.** In UC09, we train a classifier to recognize faces. This can be used to identify frequent customers or enable face identification systems[6]. The data set contains a set of computer generated face images for a subset of the customers. For each customer image, we also generate a foreign key to the customer table as meta data, to enable labeling and scoring. After loading the data in the acquisition step, the names are encoded and the images are aligned and resized. During training, a pre-trained embedding (a convolutional neural network) is fine-tuned for the customer images and a logistic regression model is trained on the embedding and the names of the customers. The serving step recognizes customer images and the accuracy is scored.

**UC10 - Fraud Detection.** In UC10, we cover a fraud detection use case. For this, there are separate financial transactions in the data set, which contain transaction time, transaction amount, sender, receiver, and transaction limits for accounts. Based on a predefined set of fraudulent transactions, a logistic regression model is trained for classification. The data is disaggregated and distributed in the tables financial_account and financial_transaction, which need to be joined and cleaned. The transaction data is normalized across the full tables before training the model. In the serving stage, transactions are classified and scored using classification accuracy.

### 3.4 Benchmark Tests and Run Rules

A complete benchmark run consists of several tests, some of which are timed and part of the measured result. The following tests are run in sequential order:

1. Data Generation
2. Load Test
3. Power Training Test
4. Power Serving Test I
5. Power Serving Test II
6. Scoring Test

[6] Given the ethical implications of facial recognition technologies UC09 will most likely be adapted to use object images instead of faces in future versions of TPCs-AI.
During Data Generation all data sets are generated including training, serving, and scoring data sets. The data generation is not timed. In the Load Test the data is loaded into the system, which means the data is moved into the final position, where it will be used by the system. This could be a copy operation or an ingestion into a system including data transformations, format changes, compression, encoding, etc. The Load Test is timed and the end-to-end elapse time is part of the final metric. The Power Training Test runs the training pipelines of the ten use cases sequentially, including all preprocessing, model training, and postprocessing tasks. The end-to-end elapse time of each use case is measured. In the Power Serving Test I and Power Serving Test II tests, the serving pipeline of all use cases are run sequentially and the elapse time of each use case is measured. In the Scoring Test, the quality of the predictions of each use case serving pipeline is measured against predefined thresholds. This test also embodies part of the model validation stage that is commonly found in production AI pipelines to decide whether a recently trained model should be deployed or discarded [37]. The time of the scoring is not measured as its purpose is to measure the quality of the models created during the Power Training Test. In the Serving Throughput Test, concurrent streams of the ten serving pipelines are defined and run. Each stream consists of the sequential execution of a permutation of the serving pipelines of the ten use cases. The end-to-end time of the execution of all streams is measured.

A complete run consists of a Validation Run with Scale Factor 1, followed by the actual Benchmark Run with the scale factor selected by the user. The purpose of the Validation Run is to verify the system under test is in a good state for running the benchmark. The results of the Validation Run are compared to a predefined result set, which is part of the benchmark kit. As part of the validation, the result cardinalities and data set sizes are compared to predefined results and other sanity checks are performed.

An official TPCx-AI result mandates that modifications to the kit are done only as allowed by the benchmark specification, and that all components of the system are commercially available (including software frameworks and libraries). An exception are TPCx-AI approved compute libraries. The result of an official benchmark run is reported to the TPC in a full disclosure report, which additionally contains all hardware and software as well as all information and configurations necessary to rerun the benchmark on the SUT.

3.5 Quality Metric

One of the acceptance criteria for a benchmark run is the quality metric. The quality metric for TPCx-AI is used to measure the quality of a trained machine learning model with previously unseen data against a ground truth data set. We call this process SCORING and it is embedded into the benchmark driver. New test sponsors do not need to implement or measure the quality themselves. What is considered high or low quality is based on the particular use case and business requirements. Hence each use case has its own predefined quality metric and threshold. See Table 2 for a full reference of the different metrics, their thresholds, and an indicator showing if a certain metric should be minimized or maximized. However, since the quality metric is seen as an acceptance criteria rather than being part of the performance metric it is enough if the threshold is met, i.e., deviations are neither penalized nor rewarded. This is quite similar to the time-to-accuracy metric of MLPerf [18], in the sense that it acknowledges the fact that perfect accuracy is not always necessary or achievable. Slight improvements in accuracy might not warrant the cost from a business standpoint if they entail significant increases in training time. On the other hand, having a quality metric threshold also acknowledges the fact that faster trained AI models may have bad prediction quality, for instance due to their sensitivity to batch sizes and learning-rate values.

We chose the following metrics for the use cases:

- word_error_rate measures how many words are correctly transcribed, the lower the better, zero is a perfect score and one and above means that every word was transcribed incorrectly.
- mean_squared_log_error the Mean Squared Log Error MSLE is used mainly for regression problems and measures the error of the predicted value and the true value but is aware of the order of magnitude. An MSLE <= 1.0 means that the predicted values are of the same order as the true values on average, hence lower is better and zero is the best.
- f1_score is a combined metric that takes precision and recall into consideration, i.e., how many of a certain category are found vs. how many of the predictions in this category are correct, higher is better and one would be a perfect categorization.
- matthews_corrcoef The Matthews Correlation Coefficient MCC measures the correlation between the predicted and true values, a MCC = 1 would mean a perfect correlation, MCC = 0 would be the value of two independent random variables, and MCC = −1 would mean a perfect anti-correlation, hence higher is better.
3.6 Metric

TPCx-AI defines three primary metrics, the performance metric, \( \text{AIUCpm@SF} \); a price-performance metric, \( \$/\text{AIUCpm@SF} \); and the availability data. All three primary metrics need to be reported for an official benchmark result. \( \text{AIUCpm@SF} \) stands for AI Use cases-per-minute at scale factor SF, and it is a combination of the elapsed times in seconds of the benchmark runs. It is designed to prevent performance improvements in only small parts of the benchmark to massively impact the overall performance metric. To this end, geometric means are used to combine the separate run times. This way, relative improvements in individual parts of the training or serving pipelines have the same influence on the overall result. In contrast, an arithmetic mean would be mainly influenced by improvements in long running stages. The use of a geometric mean is in line with previously published TPC benchmarks, such as TPC-DS and TPCx-BB [9, 23] and also recently published experiments [6]. The important parts of the metric are:

- \( SF \) the user-defined scale factor, which defines the size of the overall data set and approximates the data set size in GBs,
- \( N \) the number of use cases (i.e., 10 in the current version),
- \( S \) the user-defined number of concurrent streams to run in the \textit{Serving Throughput Test},
- \( T_{LD} \) the loading factor, which is the overall time it takes to ingest the datasets into the data store used for training and serving,
- \( T_{PTT} \) the \textit{Power Training Test} factor, defined as the geometric mean of the training times \( t_i \) of all use cases \( N \) of all use cases \( N \), the number of streams in the \textit{Serving Throughput Test}.

Intuitively this leads to test sponsors maximizing the value of \( S \) until no gains are observed in the value of \( T_{TT} \), indicating compute resources are saturated.

The resulting performance metric \( \text{AIUCpm@SF} \) is defined as:

\[
\text{AIUCpm@SF} = \frac{SF \times N \times 60}{\prod_{i=1}^{N} t_i}
\]  

(1)

Elapsed times are always reported in seconds, hence the constant value 60 is introduced to match the semantics of the performance metric. Furthermore, the performance metric is always reported for the benchmarked scale factor, e.g., \( \text{AIUCpm@10} \) for scale factor 10, and results are not comparable across scaling factors. The price performance, \( \$/\text{AIUCpm@SF} \), is defined as the total system price divided by performance metric. The system price must include all software and hardware components of the deployment including operation, maintenance, and administration for one year. The system availability date specifies the date, from which on a customer can buy the deployment as it was used for the benchmark.

Besides the three primary metrics, a benchmark sponsor can also optionally report the energy metric, Watts/\( \text{AIUCpm@SF} \). The energy measurement is centrally specified for all benchmarks in the TPC-Energy specification [25].

4 DATA SET CHARACTERISTICS

In this section, we give an overview of the data set scaling characteristics. One of the main contributions of TPCx-AI is its scalable data set. For ease of use each scale factor corresponds approximately to the size of the training data set in GB. Although the official data sizes are limited to scale factors 1, 3, 10, ..., 10,000, the data generator can produce any other data size in-between.

Figure 4 shows the data size at scale factors 1, 10, 100, and 1000. It can be seen that the data size is directly proportional to the scale factor and close to 1GB/SF. Since the data set is meant to be realistic, we do not scale all tables linearly but try to make them scale realistically in relation to the number of users and products. To
this end, each table has an individual scaling function with respect to an internal scale factor and a solver is used to approximate the actual size on disk for the aforementioned internal scale factor and scale it to the official scale factor.

Figure 5 illustrates the size of the individual data tables at the same scale factors. Even though the data set as a whole grows linearly and proportional to the scale factor, the tables grow at different rates. For instance, the lineitem ten table accounts for approximately 50% of the data set at scale factor $SF = 1$ whereas at a higher scale factor $SF = 1000$ it makes up only 25% of the whole data set. As can be seen in Figure 5, the number of customer entries grows linearly, but other tables grow faster, this is most notable with failures and customer images. Table 3 gives a detailed list of the individual table size and row counts.

5 BENCHMARK IMPLEMENTATIONS

TPCx-AI is an express benchmark and comes with a full reference implementation. According to TPC rules, this implementation supersedes the specification in cases of divergence. This means alternative implementations have to do equivalent work and produce the same output as the reference implementation. The reference implementation also increases the ease of use of the benchmark. For other benchmarks, such as TPC-H or TPC-C, a test sponsor has to implement a complete driver framework, which then also has to be certified by an auditor. Unlike previous TPC benchmarks, TPCx-AI includes two reference implementations, one for single node setups and one for cluster deployments.

Regardless of the implementation each use-case has two AI processing pipelines; one for model training on a training data set, and one for serving the trained model to make predictions on a serving data set. Each pipeline includes loading, preprocessing, training or serving, and postprocessing tasks. Figure 3 illustrates UC02’s training and serving pipelines at a high level, which performs audio transcripts as described in Section 3.3. The first stage of the pipeline is data acquisition, which retrieves the data set from the data store and makes it available for further processing. In the second stage, data transformation is executed. The data set is transformed into features and potentially labels, if it is possible. The data transformation stage can include tasks such as, data cleaning, joining, aggregations, normalization, encoding, projections, or scaling. In the third stage, the model is trained. The transformed data is used to train one or several machine learning models. The last stage in the use case is the serving stage, which uses the model, as well as the transformed data set and its labels and features to make predictions.

The implemented pipelines are purposely kept simple, with a minimum number of libraries used. Furthermore, we follow best practices as outlined by the libraries and frameworks used, rather than fine tuning the use cases, in order to keep them portable and easy to maintain, as well as easy to understand as a reference. Listing 1 shows the structure of such a pipeline implementation.

```
import pandas as pd

def load_data(path):
    return pd.read_csv(path)

def preprocess(data):
    # preprocessing steps...
    return processed_data

def train(training_data, labels):
    model = train(data, labels)
    return model

def serve(model, data):
    predictions = serve(model, data)
    return predictions

def main():
    raw_data = load_data(input_path)
    (labels, data) = preprocess(raw_data)
    if stage == 'training':
        model = train(data, labels)
        dump(model, work_dir / file_name)
    if stage == 'serving':
        model = load(work_dir / file_name)
        predictions = serve(model, data)
        predictions.to_csv('predictions.csv')
```

Listing 1: Simplified Code Structure of a Single-Node Use Case

The decision for two reference implementations was made based on the observation that machine learning pipelines come in two flavors. Many applications only require limited compute resources and all data can fully fit into main memory of a single compute unit, in these cases, single node deployments are a good choice and can often be found in practice. However, there are also pipelines and data sets, for which a single computational unit is not sufficient and the computation is distributed among several compute units. To acknowledge this fact, we include a single node implementation and a distributed implementation for the benchmark.

Workload and data set are the same for both implementations, this means, we use the same type of ML models, which are trained with the same data sets in both implementations. From a benchmark point of view, we do not focus on algorithmic improvements, but rather focus on comparability across frameworks and hardware. The model complexity stays constant across scale factors, which means higher scale factors mainly test the data throughput of systems under test. On current hardware and libraries, the single node implementation is better for small scale factors, while the distributed implementation is targeted for larger scale factors, which then are executed in shared-nothing cluster setups. The overall constraints for both implementations and any future implementations are given by the specification\(^3\) and the driver harness.

Besides the use case implementations, the kit contains the specification and the user guide, as well as configuration files to adapt the execution to the system under test. Furthermore, the kit includes the driver, which is based on a set of scripts. The scripts execute the benchmark, perform measurements, and compute the benchmark results. The kit also generates a report, which can be used as a basis for the full disclosure report, required by the TPC for an official benchmark result. Finally, the kit includes the data generator and reference data sets for verification. Additional implementations can be included in the kit to support for instance new machine learning frameworks, as well as other libraries, execution environments, or hardware components. These have to be accredited by the TPC before being used for submission of official results.

\(^3\)TPCx-AI - https://www.tpc.org/tpcx-ai/default5.asp
Table 3: Table sizes and row counts

<table>
<thead>
<tr>
<th>Table</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>conversation_audio</td>
<td>26.4 MB</td>
<td>134.0 MB</td>
<td>894.0 MB</td>
<td>4.3 GB</td>
</tr>
<tr>
<td>customer_images</td>
<td>8.0 MB</td>
<td>207.2 MB</td>
<td>7.5 GB</td>
<td>209.8 GB</td>
</tr>
<tr>
<td>product_rating</td>
<td>1.8 MB</td>
<td>9.8 MB</td>
<td>63.4 MB</td>
<td>358.0 MB</td>
</tr>
<tr>
<td>customer</td>
<td>8.5 MB</td>
<td>43.1 MB</td>
<td>258.2 MB</td>
<td>1.4 GB</td>
</tr>
<tr>
<td>failures</td>
<td>8.4 MB</td>
<td>215.0 MB</td>
<td>7.7 GB</td>
<td>217.7 GB</td>
</tr>
<tr>
<td>financial_account</td>
<td>96.9 kB</td>
<td>491.6 kB</td>
<td>2.9 MB</td>
<td>15.6 MB</td>
</tr>
<tr>
<td>financial_transactions</td>
<td>487.3 MB</td>
<td>58,023,250</td>
<td>40.7 GB</td>
<td>283.2 GB</td>
</tr>
<tr>
<td>lineitem</td>
<td>426.4 MB</td>
<td>188,575,563</td>
<td>35.6 GB</td>
<td>247.8 GB</td>
</tr>
<tr>
<td>marketplace</td>
<td>12.4 MB</td>
<td>63.2 MB</td>
<td>378.8 MB</td>
<td>2.0 GB</td>
</tr>
<tr>
<td>order</td>
<td>124.9 MB</td>
<td>29,011,625</td>
<td>10.4 GB</td>
<td>72.6 GB</td>
</tr>
<tr>
<td>order_returns</td>
<td>16.2 MB</td>
<td>9,428,778</td>
<td>84,844,260</td>
<td>9.4 GB</td>
</tr>
<tr>
<td>product</td>
<td>24.0 kB</td>
<td>122.0 kB</td>
<td>731.7 kB</td>
<td>3.9 MB</td>
</tr>
<tr>
<td>Total</td>
<td>1.1 GB</td>
<td>31,027,497</td>
<td>107.4 GB</td>
<td>1.1 TB</td>
</tr>
</tbody>
</table>

Figure 5: Data size for the all data sets per scaling factor

5.1 Single-Node Implementation

The single-node implementation is tailored towards smaller scale factors were a single data set fits into memory. This setup is typically found during the development of machine learning pipelines and for prototyping. We use Python as a programming language and framework because of its widely used data science and machine learning ecosystem. Similarly, Pandas is used for most data manipulation and data transformation. We use DataFrames to transfer the data between stages. For training the machine learning pipelines, we use scikit-learn and Tensorflow and its Keras API.

Each use-case is implemented as a single command line application, which can be controlled by providing various command line arguments that specify the training or serving pipeline, the input paths, the working directory, the output directory, and hyper parameters that are allowed to be changed by the benchmark user.

The Python-based pipelines all follow a similar structure and due to the single node setup the execution does not require complex setup and tear down procedures. A single pipeline, as outlined in Listing 1, has the following steps:

1. Load the data from one or more CSV files and merge them into a single pandas DataFrame,
2. transform the DataFrame into a feature matrix and a labels vector (numpy arrays or DataFrames depending on the use-case),
3. train the machine learning algorithm on the features and labels,
4. serve the model and create prediction for features extracted from previously unseen data and write a csv file to disk.

5.2 Distributed Implementation

For the distributed implementation, we use Apache Spark as the data processing engine and its DataFrame API for data manipulation and data transformation. Apache Spark features a machine learning library with distributed implementation for many popular machine learning algorithms. We use it for many pipelines in our implementation. The deep learning models are similar to the single-node implementation and we also use the Keras API of

---

8Python - https://www.python.org/
9pandas - https://pandas.pydata.org/
10scikit-learn - https://scikit-learn.org
11Tensorflow - https://www.tensorflow.org/
12Keras - https://keras.io/
13Apache Spark - https://spark.apache.org/
Tensorflow for the distributed implementation. To train the models a data parallel approach is used, i.e., the same model is trained on different partitions of the training data set and the gradients are gathered after each step. This approach in particular is called allgather/allreduce and implemented by Horovod [33].

However, the computational model is completely different. While the execution for the single-node implementation is procedural and imperative the distributed implementation is much more declarative. We use the `DataFrame-API` of Apache Spark to create an execution graph, which is then optimized and executed by the Spark runtime. The `DataFrames` used in the context of the distributed implementation are different from the single-node version. Using `DataFrames` enables working with data sets that do not fit into memory and can be stored in a distributed file system.

The structure of the Spark pipelines is similar to the Python-based pipelines and has the same steps of loading, data transformation, model training, and serving. Unlike the Python-based execution, the Spark engine can use lazy execution within a job and does not have to materialize all intermediate steps, such as data transformations before model training.

### 6.1 Single Node - Python-based

We test Scale Factors 1, 5, 10, and 15 using the Python-based kit. For the `Serving Throughput Test` we use 2 streams. We were not able to run larger scale factors in Python, because of memory constraints of our testing setup and limitations in the libraries [15] that have been used. The performance metrics of the runs can be seen in Table 5. Officially, the numbers are not comparable across scale factors, which also makes sense, since not all tables scale linearly. However, as can be seen in the table, the partial metrics across different scale factors grow with the scale factor, but not linearly. All of the individual metrics are less than a factor of 10 larger in SF 10 than in SF 1. This results in a improved primary performance metric for SF 10 compared to SF 1. This is due to the structure of the metric, which incorporates the scaling factor in the final measure. Just looking at the performance metric one could conclude that with the hardware and software being used for these experiments that the single-node implementation performs best at scaling factor $SF = 5$ or in other words on a 5GB data set.

Looking at the quality metrics (see Table 4) it can be observed that each benchmark run at each scaling factor was valid since the thresholds were met for all use cases. Furthermore, it can seen that some use cases improve at higher scaling factors, some stay more or less constant and others even worsen slightly. Even though the first case is to be desired in some cases, it might be that the model was actually overfitting at low scaling factors and is becoming more generalized with more training data.

In Figure 6 the individual run times for each use case in the Python-based execution with scale factor 1 on our server is shown. In the plot the average run time for each use case is displayed for the `Serving Throughput Test`. It can be seen that the serving time is similar for the `Power Serving Test` and the `Serving Throughput Test`. However, in the metric, we get a better value, showing that multiple streams enable more parallelism on the server. The training is always much more costly than the serving stages. Additionally, we can see that the run times of the use case differ in more than two orders of magnitude, which is to be expected in real workloads.

### 6.2 Multi Node - Apache Spark-based

For our multi node experiments, we use an 11 node setup (1 main node and 10 worker nodes). We execute Scale Factors 1, 10, 100, 200, 300, and 400 on our cluster and use two streams in the `Serving Throughput Test`. The performance metrics of the runs can be seen in Table 5. Interestingly, the performance numbers for smaller scale factors in the Spark-based experiments setup are slower than in the Python-based runs. This is due to the higher startup times of Spark. Additionally, we can see that the run times for Scale Factor 10 are similar to the run times of Scale Factor 1, which also shows that small scale factors are not properly utilizing the cluster. For Scale Factor 100, some of the stages seem to be performing more work and are taking considerably more time than in the smaller scale

---

[15] see https://github.com/pandas-dev/pandas/pull/45084 for an issue that affected uc08 and has since been fixed.
Table 4: Quality metric for single- and multi-node experiments

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UC01</td>
<td>N/A</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>UC02</td>
<td>WER</td>
<td>0.31</td>
<td>0.35</td>
<td>0.29</td>
<td>0.18</td>
<td>0.88</td>
<td>0.93</td>
<td>0.32</td>
<td>0.26</td>
<td>0.22</td>
<td>0.23</td>
</tr>
<tr>
<td>UC03</td>
<td>MSLE</td>
<td>4.60</td>
<td>3.34</td>
<td>3.62</td>
<td>3.30</td>
<td>5.73</td>
<td>3.70</td>
<td>3.87</td>
<td>4.25</td>
<td>4.36</td>
<td>4.48</td>
</tr>
<tr>
<td>UC04</td>
<td>F1</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
<td>0.69</td>
<td>0.70</td>
<td>0.69</td>
<td>0.70</td>
<td>0.70</td>
<td>0.70</td>
</tr>
<tr>
<td>UC05</td>
<td>MSLE</td>
<td>0.01</td>
<td>0.04</td>
<td>0.02</td>
<td>0.02</td>
<td>0.30</td>
<td>0.21</td>
<td>0.04</td>
<td>0.03</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>UC06</td>
<td>MCC</td>
<td>0.46</td>
<td>0.53</td>
<td>0.54</td>
<td>0.54</td>
<td>0.23</td>
<td>0.23</td>
<td>0.20</td>
<td>0.23</td>
<td>0.22</td>
<td>0.21</td>
</tr>
<tr>
<td>UC07</td>
<td>MAE</td>
<td>0.89</td>
<td>0.98</td>
<td>1.02</td>
<td>1.01</td>
<td>1.71</td>
<td>1.62</td>
<td>1.31</td>
<td>1.44</td>
<td>1.43</td>
<td>1.47</td>
</tr>
<tr>
<td>UC08</td>
<td>Acc</td>
<td>0.71</td>
<td>0.73</td>
<td>0.74</td>
<td>0.74</td>
<td>0.72</td>
<td>0.73</td>
<td>0.76</td>
<td>0.75</td>
<td>0.77</td>
<td>0.72</td>
</tr>
<tr>
<td>UC09</td>
<td>Acc</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>UC10</td>
<td>Acc</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
<td>0.82</td>
</tr>
</tbody>
</table>

Table 5: Performance metric for single-node and multi-node experiments

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>𝑇𝐿𝐷</td>
<td>2.38</td>
<td>4.99</td>
<td>15.19</td>
<td>14.63</td>
<td>94.19</td>
<td>138.12</td>
<td>797.76</td>
<td>1729.97</td>
<td>2735.43</td>
<td>5993.25</td>
</tr>
<tr>
<td>𝑇𝑃𝑇𝑇</td>
<td>91.18</td>
<td>352.61</td>
<td>905.16</td>
<td>1432.03</td>
<td>130.02</td>
<td>153.21</td>
<td>328.58</td>
<td>441.98</td>
<td>588.84</td>
<td>761.54</td>
</tr>
<tr>
<td>𝑇𝑃𝑆𝑇</td>
<td>9.92</td>
<td>23.11</td>
<td>52.39</td>
<td>65.33</td>
<td>58.59</td>
<td>64.07</td>
<td>77.70</td>
<td>91.27</td>
<td>96.30</td>
<td>105.02</td>
</tr>
<tr>
<td>𝑇𝑇𝑇𝑇</td>
<td>14.26</td>
<td>38.04</td>
<td>97.62</td>
<td>165.86</td>
<td>35.70</td>
<td>41.66</td>
<td>58.19</td>
<td>84.34</td>
<td>102.42</td>
<td>124.46</td>
</tr>
<tr>
<td>𝐴𝐼𝑈𝐶𝑝𝑚@𝑆𝐹</td>
<td>45.28</td>
<td>85.10</td>
<td>65.51</td>
<td>73.31</td>
<td>8.43</td>
<td>69.20</td>
<td>323.37</td>
<td>433.22</td>
<td>506.99</td>
<td>485.61</td>
</tr>
</tbody>
</table>

Table 6: Performance metrics of two official TPCx-AI results

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Single-Node (Python) SF=10</th>
<th>Spark SF=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>𝑇𝐿𝐷</td>
<td>4.53</td>
<td>4,405.47</td>
</tr>
<tr>
<td>𝑇𝑃𝑇𝑇</td>
<td>314.80</td>
<td>981.90</td>
</tr>
<tr>
<td>𝑇𝑃𝑆𝑇</td>
<td>26.89</td>
<td>128.21</td>
</tr>
<tr>
<td>𝑇𝑇𝑇𝑇</td>
<td>4.69</td>
<td>110.67</td>
</tr>
<tr>
<td>𝐴𝐼𝑈𝐶𝑝𝑚@𝑆𝐹</td>
<td>291.35</td>
<td>1,205.43</td>
</tr>
</tbody>
</table>

6.3 Official TPCx-AI Results

As of June 2023, 17 results have been published on the TPC website. To give a comparison of our experiments to real benchmark results, we briefly discuss the first two results here.

The first result was submitted by the Telecommunications Technology Association (TTA) in April 2022. It is an SF 1000 run (1 TB) with 10 streams executed on Apache Spark. The system under test consists of 3 servers with each 2x Intel Xeon Platinum 8380 CPU and 2,048GB memory. In total the system has 6 processors with 240 cores and 480 threads, and 6,144GB memory. The total disk space is 136TB. The primary metrics of the system are:

- **Performance**: 1,205.43 AIUCpm@1000
- **Price/Performance**: ₩378,912.09 KRW/AIUCpm@1000 ($290 USD/AIUCpm@1000)
- **Availability date**: April 18, 2022
- **Total System Cost**: ₩456,752,000 KRW ($350,000 USD)

The second result was submitted by Nettrix Information Industry in May 2022. It is an SF 10 run with 100 streams using the Python-based implementation. The system under test is a one-server system consisting of 2x Intel Xeon Gold 6346 CPU (2 processors with 32 cores and 64 threads) with 512GB memory and a total disk space of 12TB.

- **Performance**: 291.35 AIUCpm@10
- **Price/Performance**: $84.26 USD/AIUCpm@10
- **Availability date**: May 31, 2022
- **Total System Cost**: $24,548 USD

16TPCx-AI result webpage - https://www.tpc.org/tpcx-ai/results5.asp
17ID 12204211: TPCx-AI Result Highlights TTA KR580S2 - https://www.tpc.org/5401
18https://www.tta.or.kr/
20https://www.nettrix.com.cn

Factors. Overall, the Spark cluster can execute larger scale factors and get better performance numbers for these scale factors.

Similar to the performance metric the quality metric also performs worse at low scale factors than for the single-node implementation but improves at the high and very high scale factors for some use cases. An interesting observation is that for UC02 and UC03 the default parameters will not produce a valid benchmark run due to the quality metrics exceeding the thresholds, the values highlighted in red in Table 4. The reason is, especially for UC02, that the data is partitioned, a model is trained on each partition, and all partition models are then merged. When the partitions are too small then these partition-based models are not able to perform well. This can be changed by reducing the parallelism.

In Figure 7 the individual test run times of each use case for Scale Factor 10 are shown. Again, we report the average run time for the Serving Throughput Test. Overall, the performance characteristics are similar to the Python experiment in Figure 6. An outlier is UC06, which is the slowest use case in Spark, while it is fast in Python. However, the failures table grows faster than the scaling factor, meaning in the larger scale factor disproportionately more work needs to be done for UC06.
The detailed result metrics of both benchmark runs are presented in Table 6. All details are available on the TPC website. Since then there have been additional results published by Dell Technologies, Hewlett Packard Enterprise, and Transwarp Technology in different scale factors, ranging from 3 to 3000.

7 FUTURE USE CASES
In addition to the ten published use cases, we created a set of six new use cases to be added to the workload in future versions. The list of new use cases is shown in Table 7. This set of use cases focuses more on deep learning based models. These use cases add new tables and attributes to the data set and will, therefore, change the data set scaling behavior. The use cases are not available publicly yet. In the following sections, we give a brief introduction to each new use case.

UC11/UC13 - Sentiment Analysis ML/DL In these use cases the sentiment of a product review is predicted either by using traditional ML or DL techniques, i.e., random forest model or a Recurrent Neural Network. In the preprocessing stage, the data is loaded and the classifier is trained using star ratings as labels.

UC12 - Product Description Translation In UC12, product descriptions are translated from English to German. A transformer model is used to translate the sentences in product descriptions, after it has been trained on example sentences in both languages.

UC14 - Credit Scoring UC14 predicts credit ratings for merchants based on existing loans using a multi-layer perceptron model. The preprocessing stage joins multiple tables and including new tables for merchants. The values are normalized and missing values imputed. In the serving stage, ratings for merchants applying for loans are predicted.

UC15 - Predict Retail Location In UC15, locations for new stores are predicted based on customer demographics. In the training stage, a decision tree is trained, which is then used to perform regression for new store locations.

UC16 - Click-Through-Rate Prediction UC16 predicts click conversion to purchases using a deep learning recommendation model. Click logs are used and converted in a click log model. In the serving stage, click sessions are classified by their conversion probability.

8 CONCLUSION AND FUTURE WORK
In this paper, we present TPCx-AI, the Transaction Processing Performance Council’s new industry standard benchmark for artificial intelligence and machine learning systems. TPCx-AI is modeled on a retail scenario and comprises ten realistic use cases —covering both deep learning and traditional machine learning algorithms —each of which is implemented as an end-to-end AI processing pipeline. The pipelines consist of data ingestion, preprocessing, training, serving, and postprocessing as well as model update. Since TPCx-AI is a TPC Express benchmark, it comes with a fully implemented kit, which is ready to run on common ML/AI hardware. Besides a single node Python-based implementation, it also features a distributed Apache Spark-based implementation. TPCx-AI has already received several independent officially audited result submissions after being officially standardized in September 2021, underlining the interest in industry for the benchmark.

In future work, we want to separate preprocessing and postprocessing from the training and serving stages into independent stages and, furthermore, enable a more fine grained optimization of pipelines as well as adding continuously evolving pipelines [37]. Additionally, we hope to include the new use cases and more system implementations in future versions of the benchmark. A further extension can target other challenges found in production AI pipelines, such as model maintenance or update [2].

ACKNOWLEDGMENTS
This work was partially funded by the German Research Foundation (ref. 414984028), and the European Union’s Horizon 2020 research and innovation programme (ref. 957407).