TQwerE: Transformer-Based SQL Query Executor

Nir Regev, Asaf Shabtai and Lior Rokach

Dept. of Software and Information Systems Engineering
Ben-Gurion University of the Negev
Be'er Sheva, Israel

Abstract

Recent developments in large language models (LLMs) trained on large-scale unstructured textual data have produced high-achieving models. However, it remains a challenge to train an LLM on vast structured (tabular) data for the task of understanding the information captured in the data and answering questions regarding the data. We propose a novel method - TQwerE, for approximating SQL aggregated queries' results over large data sets. Our main focus was to reduce query latency and incurred costs. Moreover, since we focus on large data sets, majority of models that scan raw data are not applicable. Instead, our method fine tunes Jurassic-2 to learn the relations between aggregated SQL queries and their results without referring directly to the underlying raw data. We demonstrate TQwerE's ability to approximate aggregated queries with state-of-the-art accuracy and speed. We evaluated TQwerE on twelve datasets, and our results demonstrated its superiority to both the state-of-the-art methods.

Keywords

LMM, SQL , AQP, LSTM

1 Introduction

Pre-trained large language models (LLMs) such as GPT ([4]), BERT ([9]), and BART ([21]) have shown great success on a wide range of natural language processing (NLP) tasks. LLMs are fed a large amount of unstructured textual data and trained to predict a textual token within a sequence of tokens. Once pre-trained, these models have demonstrated astounding capabilities in understanding natural language (NL) sentences. Motivated by this, researchers have shifted their attention from textual data to structured tabular data ([15], [36], [38], [34], [8], [30]). Existing pre-training approaches were originally designed for unstructured data and are not well suited for tabular data, which unlike NL data commonly contains rich and meaningful structural information. To adapt existing pre-training techniques for use on tabular data, three main hurdles need to be overcome: (1) designing an efficient pretraining task that is suitable for tabular data, (2) building a fine-tuning (training) set and (3) performing the fine-tuning training task cost-effectively and within a reasonable amount of time. To address these challenges, many studies have collected tuples of data consisting of NL sentences and tables for the pre-training corpus since downstream tasks often involve joint reasoning on both free-form NL sentences and tables. This was done by scraping tables and their surrounding NL sentences from the web ([15], [36], [7]) or by synthesizing NL sentences on available tables ([38], [30]).

In this paper, we present TQwerE, a transformer-based SQL aggregated query predictor. The motivation for this research is to significantly reduce query latency, cutting computational costs and keeping model predictions accuracy high. This approach, known as AQP (Approximate query processing) improves performance, particularly for large-scale data, by leveraging predictive capabilities, making it ideal for exploratory analysis, real-time dashboards, and cost sensitive workloads.

To backup this motivation let us take a use-case where a group of 10 analysts explore the relationship between life expectancy and socioeconomic factors (such as income). A cloud-based storage with 20 data attributes is available for this task populates the related data for every adult in the world (5.3 billion adults globally) and takes about 5.3 TB (20 fields×50 bytes= 1KB per adult). To calculate the required number of queries for such research we will assume that only 3 continuous factors are taken into the analysis: (1) income, (2) expenditure (spending on goods and services) and debt (loans and mortgages) where each factor is binned into 10 levels. The overall combinations of these factor will surge to 1000, which is approximately the number of queries required for a single analyst to execute. If every analyst explores different set of factors, we can assume that approximately 10,000 queries targeted to calculate the average life expectancy grouped by each city (10,000 cities globally) will be needed for this research with each scanning 5.3 TB of data. For simplicity we assume no storage cost, and only compute cost will be considered. This will incur compute cost of 83 $ for a single query (based on AWS Lambda: $0.00001667 per GB processed). Overall costs for all queries are 830,000$. Scanning 5TB is estimated to take 11.5 hours (based on a high-speed connection 1 Gb per second). To summarize, the described use-case takes significant resources in terms of costs and time to carry out the data exploration tasks for such research. Naturally, the above pipeline is not feasible for most companies, this is the reason why analysts will sample a fraction of the data and risk sampling biases that can skew research results. A different approach to run aggregated SQL queries is described in The AQP method [29] and may serve as a perfect fit to the described use-case. Although the AQP approach may introduces slight inaccuracies, the trade-off is acceptable for use cases where cost, speed and efficiency are prioritized over exact precision. In this research we specifically focus on data exploration which is a common task for data scientist and analysts. Data exploration on large datasets can be resource-intensive, requiring significant computational power for querying, aggregating, and visualizing vast amounts of data. This process may incur high costs in terms of processing time, cloud storage, and compute resources, especially when repeated queries or full dataset scans are involved. Additionally, slow query performance can hinder analysts' productivity, delaying insights and decision-making. These summaries the justification for a light ML based SQL query executor.

With this motivation, we propose a novel method - TQwerE, for approximating SQL aggregated queries' results over large data sets. This method is for the goal of data exploration usually done by data scientists prior to building a ML model. Our main goal was to reduce query latency and cut down incurred costs that are often pose a limitation when data scientists and analysts explore large data set. In addition, since we focus on large data sets, majority of ML models that provide AQP scan raw data and therefore not applicable in our case. Instead, our method fine tunes Jurassic-2 to learn the (only) relations between aggregated SQL queries and their results without referring directly to the underlying raw data. We show TQwerE's ability to approximate aggregated queries with improved accuracy and speed. TQwerE addresses this task by first constructing a diverse SQL queries training set (referred to as a fine-tuning set) and then performing a fine-tuning task. We evaluated TQwerE on twelve datasets, and our results demonstrated its superiority to both the state-of-the-art approximate query processing (AQP) method and Tapas - a LLM developed by Google for question answering tasks over tabular data. TQwerE is a pipeline for finetuning a LLM foundation model to execute aggregated SQL queries in an AQP approach. TQwerE consists of four main steps:

1. generating a set of SQL queries with a reference to a dataset, which we to refer as a fine-tuning set,
2. obtaining the queries' ground-truth label (query result),
3. fine-tuning the foundation model on the fine-tuning set, and
4. validating the fine-tuning process by executing validation set queries and comparing the results (as predicted by the transformer model) to the ground truth labels.

In our evaluation we gathered the results of running the validation set queries with each of the benchmark methods: (1)TQwerE, (2) AQP method ([29] trained on the same validation set queries, (3) TAPAS - a LLM designed for answering natural language questions over tabular data. This method was fine-tuned the same way TQwerE was trained (on the same validation set queries), and finally (4) a statistical smart sampling (referred to as SS in the remainder of the paper) method ([25]).The latter is a statistical method which derives representative data samples from the original datasets and runs the testing set SQL queries against the data samples. We compared the query results accuracy (measure by weighted MAPE as explained ahead) and query run-time latency. Since TQwerE supports aggregated queries (which returns scalars) an appropriate metric to measure accuracy is weighted MAPE - mean absolute percentage difference between ground truth and query result weighted by the normalized rows count of the query. We used weighted MAPE because it measures the percentage difference which enables us to compare performance across diverse datasets with different distributions. To measure query latency, we calculated the elapsed runtime of each query for each method. TQwerE's accuracy results were found to be superior to those of the AQP method and the SS method on six of the nine open-source, and the TQwerE query latency were equivalent to the AQP method and better than TAPAS.

The contributions of this paper are as follows:

- We introduce TQwerE: a novel pipeline for fine-tuning an LLM to estimate SQL queries results based on a tabular dataset.
- We demonstrate the proposed method's efficiency and accuracy in providing SQL query estimates which are comparable to those of the state-of-the-art method [29] and a statistical smart sampling method ([25]).
- We have made our code and related training sets publicly available at: https://anonymous.4open.science/r/aqp_jurassic-5B0E.

2 Related Works

In the following sections we will introduce some of the work done so far in the specific domain of tabular Data understanding and tabular question answering (QA).

2.1 Machine Learning on Tabular Data

As deep learning methods have been shown to be efficient on tasks based on unstructured data in both the text and image domains, there have been recent studies for learning tabular data. Tabular data tasks have largely revolved around the prediction of masked table cells or the identification or correction of corrupted cells ([33], [37], [1], [6]). Other studies have used differentiable trees, which combine the advantages of tree ensembles with the gradient-based optimization of NNs ([19], [28]). However, several studies ([32], [3], [13]) found that gradient-boosted tree ensembles like XGBoost ([5]) and LightGBM ([18]) systematically perform better these deep learning methods, even with proper fine-tuning and regularization ([17]). In [20], the authors proposed a method for predicting table cells which used a semi-supervised setting, however there was a need to perform a series of supplementary supervised tasks on the same table, which can pose a limitation. In addition, the authors explored few-shot classification for medical diagnosis, using 4-200 labelled examples, however they did not leverage the potential of pre-trained LLMs. In [16], the authors introduced TabPFN, a Bayesian NN pre-trained on synthetic tabular data, which was shown to outperform gradient boosted trees in their comprehensive evaluation.

2.2 Large Language Models for Tabular Data

In an alternative approach for learning tabular data relations LLMs' NL capabilities are leveraged. In [36], an LLM was used to semantically parse queries on tabular data. Language models' ability to perform entity matching on tabular data, i.e., determine if two rows refer to the same object, has also been explored [22]. In [14], the authors studied data enrichment by linking each row in a table with additional unstructured text (e.g., from Wikipedia), from which the authors generated additional features using a language model. This configuration requires named entities (e.g., celebrities, universities), which makes this method complex. In another study [2], the authors explored two healthcare datasets and used a language model to generate feature embeddings, which they fed to classifiers like gradient boosted trees. This study- [9] [27], The use of a BERT-style language model for tabular data cleaning tasks was evaluated with GPT-3 ([39]), and the authors discovered that it frequently outperformed state of-the-art methods with 10 labelled examples. In [3], the authors introduced an LLM method for producing lifelike tabular data. The proposed method required serializing the tabular data into a set of serialized strings, a process that increased the complexity of the data preparation task. In another study [36], a column data type was transformed into a serialized string. Only [2] examined different serialization variants, however in this case, this was done in the context of deriving feature embeddings from BERT-style language models. The LIFT method introduced by [10] most closely aligns with our work. The authors evaluated the capabilities of fine-tuned GPT-3 and GPT-J models for regression and classification on synthetic, tabular, and vision data. They also examined sample efficiency and considered different static serialization templates, assessing the effect of including column names in the input.

In this work, we focus on the publicly available Jurassic LLM to learn an SQL approximation task on nine publicly available datasets. This task mainly consists of a fine-tuning process on a generated set of SQL queries and evaluating the method's ability to approximate the queries' results with respect to their data true label (query result).

2.3 Tabular Data Pre-Training

The tabular data task typically includes key factors like the pre-training data corpus and task. In terms of the pre-training corpus, most prior studies crawled tabular data and their associated surrounding NL texts and performed pre-training to modify the foundation model. These studies either extracted information from an extensive selection of tables and their NL sentence contexts ([36]; [15]), leveraged parallel human-annotated NL-table datasets for pre-training ([7]; [38]), or synthetically generated an NL-table corpus using human experts' templates ([38]; [11]).

Finally, TAPAS (Tabular Pretrained Autoregressive Sequence) is a transformer-based language model designed specifically for learning from and reasoning about tabular data. Developed as an extension of the BERT model, TAPAS incorporates a novel approach to align natural language processing with tabular data structures. Unlike traditional LLMs, TAPAS is trained on datasets where input queries are paired with tabular information, enabling it to handle tasks such as table-based question answering and tabular reasoning. In addition, the model has been shown to perform well on complex tasks requiring multi-row reasoning, numerical computations, and aggregation, which are challenging for conventional LLMs. This is the reason why we chose TAPAS as a fair benchmark method for comparison with TQwerE.

2.4 Joint Understanding on Table and Text

Several studies ([23], [40], [30]) employed a semantic parsing task, in which reinforcement learning was used to optimize semantic parsers on tables. Although these parsers were shown to produce logic forms (e.g., SQL), they pose difficulty in training due to the large search space and presence of spurious programs ([12]). Recent advancements in tabular data understanding have opened another promising avenue of research ([26]; [15]) aimed at answering NL sentences without logical forms. In this case, the answers are predicted by selecting table cell values and applying an aggregation operator to them. While these models can be easily trained, their ability to construct complex SQL is limited. For example, it is difficult to support compound aggregation operators such as max(Year) min(Year). For table-based fact verification (TableFV), prior studies typically utilized specialized architectures with limited scalability ([34]; [31]; [35]). For example, [41] leveraged a graph construction mechanism, semantic parser, and semantic composition model to capture the connections among an NL sentence and a table. While the approach works well for TableFV, it is not easily applied to other table-related tasks. What sets our approach apart from the approaches used in prior work is that we utilize LLMs to query tabular data directly, without the need to learn it directly. This makes the fine-tuning task faster and more focused.

3 PROPOSED METHOD

In this section, we describe our proposed method: TQwerE - a transformer-based SQL query executor. Our method relies on the idea that instead of having a ML model learn the raw data set, it learns the relations between aggregated SQL queries and their results (referred as labels). This approach is well explained in [29] and is expanded to support data exploration tasks where analytical (aggregated) queries are required to solve a research question as demonstrated in section 3.4. TQwerE pipeline consists of the following steps:

1. Constructing SQL queries tailored to each dataset.
2. Obtaining the queries' ground-truth labels.
3. Fine-tuning TQwerE on the training (referred as fine-tuning set) set of SQL queries.
4. Validating TQwerE by comparing the validation set queries predicted results to the ground-truth labels.

A detailed description of these steps is provided below. The pipeline was executed on 12 datasets (see Section 4) in our experiments.

3.1 Query-Related Notations

The first step begins by defining the query structure. As an example, assume a 'life_exp' table that includes data on a adults age from different cities and gender. The table includes the following attributes: 'income', 'expenditure' (annual spending on goods), 'gender' , 'city' and 'age'. Our method is designed to generate queries that conform to a query template defined by the following:

- $attr^{(c)}$ – denotes a continuous data attribute in the dataset (e.g., 'income'). – $attr^{(n)}$ – denotes a nominal data attribute in the dataset (e.g., 'city' 'gender').
- $A = \{a_1, a_2, ...\}$ – denotes the set of optional aggregation functions (e.g., avg, count).
- $a_i(attr)$ – denotes an aggregation function $a_i \in A$ that is applied on valid attribute $attr$ (either $attr^{(c)}$ or $attr^{(n)}$) in a 'SELECT' query clause (e.g., avg('age'), max('age'), or count()).
- $between_{attr^{(c)}}(l,u)$ – a 'between' constraint argument defined on a continuous data attribute $attr^{(c)}$, where l is a lower bound and u is an upper bound on the values of $attr^{(c)}$.

- $in_{attr^{(n)}}(v_k)$ – an ‘in’ constraint argument defined on a nominal data attribute $attr^{(n)}$, where (v_k) is a single possible member of $attr^{(n)}$.

3.2 Multifaceted ‘Group By’ Queries with Multiple Aggregation Functions

Our method supports queries with the ‘Group by’ clause on multiple $attr^{(n)}$. While this enables flexibility and scalability in exploring and analysing large datasets, it poses the following two challenges: (1) the need to learn different data distributions (characterized by the aggregation functions), and (2) difficulty in learning an output which may have varying dimensions. The latter stems from the fact that a ‘Group by’ query can return a table of one or more rows, as shown in the example in Table 1, which presents the result for the following query:

```
SELECT city, gender,avg(sales)
FROM life_exp le RIGHT OUTER JOIN geo_country gc
ON le.city = gc.c_code WHERE
income between (10000 and 20000) AND expenditure
between (50000 and 60000)
GROUP BY city, gender
```

To expedite the fine-tuning task, we transform each ‘Group by’ query into multiple ‘flat’ queries with a single aggregation function. By definition, these queries return one scalar. This way, the LLM learns to output a single numeric result. From the user perspective this is transparent, as our method includes a wrapper that includes this query transformation: (1) transforming GROUP BY queries to multiple ’flat’ queries (2) predicting the ’flat’ queries results and (3) constructing a table of predicted results compatible to the GROUP BY query.

Table 1: An example of a ‘Group by’ result set.

city	gender	avg(age)	max(age)
New York	male	51.8	92.5
London	male	48.2	89.5
New York	female	52.9	94.2
London	female	49.1	96.1

3.3 Synthetic Query Generation

The objective of this step is to produce a curated set of aggregated SQL queries q_i that represents the raw data.

3.3.1 Query structure.

The query structure consists of: (1) the ’SELECT’ clause parameters, (2) the filter template (i.e., the WHERE clause parameters), and (3) the name(s) of the table(s).

In this step, a list of aggregation functions is coupled with a list of target data attributes. Then, our method builds a ’SELECT’ clause consisting of the selected aggregation functions, which are applied on a set of the target data attributes $\{a_i(attr_j)\}$.

Aggregation functions can be applied on continuous data attributes, however the only aggregation functions that can be applied on nominal attributes are *'count'* and *'countDistinct.'* In our example, assuming the domain expert chooses to apply all of the aggregation functions A on all valid $\{a_i(attr_j)\}$, the result is the following ’SELECT’ clause:

```
SELECT avg(age), max(age)
```

Each $\{a_i(attr_j)\}$ will have a designated LLM fine-tuning model variant which will be fine-tuned to learn its unique distribution. This means that the fine-tuning set will be duplicated for each $\{a_i(attr_j)\}$ and learned separately. In our example, the first training set will consist of queries with avg(sales) in the 'SELECT' clause, the second training set will consist of queries with MEDIAN(sales), and so on.

Next, a filter template, which includes the list of continuous data attributes $attr^{(c)}$ and nominal data attributes $attr^{(n)}$, is defined. Then, for each defined *query template*, our method generates a set of *query instances* as follows.

3.3.2 Generating filters.

Generating filters. In this step, the method generates a rich set of filters applied on (1) continuous data attributes $attr^{(c)}$, and (2) nominal data attributes $attr^{(n)}$ in the following manner. In a preceding step, the datasets are re-sampled with replacement (bootstrapping) to extract 30 iterations of 10% of the samples. From these samples, we estimate the dataset's attribute boundaries (min, max, 25%, 50%, and 75% percentiles) and attribute members (in cases in which the attributes are categorical attributes). This step is described ahead in more details:

1. For each $attr^{(c)}$, Our methods calculate the intervals defined by the minimum value, the first quartile (25%), the median, the third quartile (75%), and the maximum value (four intervals). In order to select the lower and upper bounds of a continuous attribute constraint $between_{attr^{(c)}}(l,u)$, we select two intervals randomly. Then, from each selected interval, we randomly choose a value sampled from a uniform distribution. This process results in two numeric values which form a filter, such that the smaller value will define the lower bound, and the larger value will define the upper bound. For instance, assume that the *income* continuous attribute values span from 10000 to 100000. Given those values, minimum=10000, 25% quartile=20000, median=40000, the 75% quartile=70000, and maximum=10000, and assuming that the selected intervals are [10000,20000] and [70000,100000], the continuous constraint might take the values $between_{income}(10125,81590)$.
2. To construct a nominal filter, our method uses an "IN" constraint argument defined on a nominal data attribute $attr^{(n)}$, filtered by v_k, which is a possible member of $attr^{(n)}$. To determine which member to use in each filter, the method constructs a 'Group by' term on the nominal attribute. Once the query is executed against the dataset, the method systematically extracts all possible combinations of members that exist in the result set and constructs a nominal filter for each combination. In our example, this is one possible combination of the members for *city* and *gender*: $\{incity('NewYork'), ingender('Male')\}$.
3. Finally, each combination of nominal filters is paired with each of the continuous filters to form a query filter, for example,

   $\{between_{income}(10125,81590),\ in_{city}('NewYork'),_{ingender}('Male')\}$.

3.3.3 Generating 'JOIN' clause.

In this step, a join clause is added to the query. TQwerE supports 'INNER' and 'OUTER' joins (left and right). The join key is configured within in the schema configuration files and is taken to build the 'JOIN' syntax in the following format:

$\{join\_type_{left\_table}(left\_key), right\_table(right\_key)\}$
where join_type can contain the values: (1) INNER, (2) LEFT, (3) RIGHT
and (4) CROSS (for the last option key is not relevant) e.g.

```
life_exp ls RIGHT OUTER JOIN geo_country gc ON ls.city =
gc.c_code
```

3.4 Life expectancy use-case.

Life expectancy use-case - data exploration queries We will base our example on a use-case where group of 10 analysts explore the relationship between life expectancy and socioeconomic factors (such as income). A cloud-based storage with 20 data attributes is available for this task populates the related data for every adult in the world (5.3 billion adults globally) and takes about 5.3 TB (20 fields×50 bytes= 1KB per adult). Assuming each analyst explores 3 different continuous factors, each will execute 1000 queries and overall 10,000 queries. Each query is targeted to calculate the average life expectancy grouped by each city (10,000 cities globally). Each query will incur compute cost of 83 $ for a single query (based on AWS Lambda: $0.00001667 per GB processed) and overall of 830,000$ for all queries (see detailed calculated in 1). From latency perspective, scanning 5TB of data is estimated to take 11.5 hours (based on a high-speed connection 1 Gb per second). Each query may look like the following:

```
SELECT city, avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 20000 AND
expenditure between 50000 and 60000 AND debt between
100000 and 20000
GROUP BY city
```

3.5 Acquiring Ground Truth for the training Set

As the objective of the second step is to build a supervised fine-tuning set, the proposed method needs to acquire the ground-truth labels (query results). Therefore, our method runs the set of generated queries *Q* over the data source. We use the Python pandasql library to acquire the ground-truth labels.

3.6 Fine-tuning LLM

In the last step, our method takes a foundation LLM and runs a fine-tuning task on the fine-tuning (a 10% sample derived from the training set) set described above. General pre-trained transformer (GPT) LLMs have been proven to efficiently learn complex context relations in sequential data using their multiple attention heads mechanism, which was our initial motivation for selecting this architecture [24]. The fine-tuning process consists of:

1. generating SQL queries tailored for each dataset,
2. retrieving the ground-truth labels (by running the query on the database),
3. constructing a CSV file containing <query, query_result_label>

(which corresponds to the "prompt" and "completion" CVS columns),

4. running the fine-tuning service for AI21labs Jurassic-2 model ([24]) andupon completion of this
5. Approximate query results from fine_tuned Jurassic LLM and calculate accuracy metrics on the validation set.

Importantly, TQwerE requires less training time than training the AQP method (LSTM), since the Jurassic LLM foundation model is necessitating a short finetuning process as opposed to the AQP method, which requires training from scratch. In addition, our method does not require encoding, which is required by the AQP method, and as a result it uses the internal encoding learned within the LLM architecture.

3.7 Data changes scenario

To handle new data coming into the data base, we suggest building a hybrid fine-tuning set of SQL queries that will scan of both newly inserted data records and "old" data records which are already learned by the model. This is described in the following method:

1. Time reference - mark data records with insert date-time attribute referredas "insert_dt" and model training date-time refereed as "model_train_dt".
2. Small fine-tuning set - Using the above-mentioned date-time attributes, builda relatively small fine-tuning set of SQL queries that will span on both newly inserted records and existing records.
3. Query split - split each query to two queries according to the"model_train_dt":
   (a) "model_query" - this query will impute a filter which is less than"model_train_dt". Acquire label for these queries by running a query against the existing model.
   (b) "data_query" - this query will impute a filter which is equal or greaterthan "model_train_dt". Acquire label for these queries by running a query against the data base.
4. Query merge - "model_query" and "data_query" results according to aggregation function logic: If "Sum" function is used, sum results of the queries, if "Average" function is used, calculate weighted (by the count of records from each query) average, if "Count" function is used, sum up the counts and if "Min/Max" is used take one of the values from the 2 queries according to the function logic.
5. Fine-tuning - Run fine-tuning task on the merged SQL queries set.

An illustration of this method with an "avg" aggregation function is described in figure 3.

4 EVALUATION

We compared our method's performance to that of two other types of models:

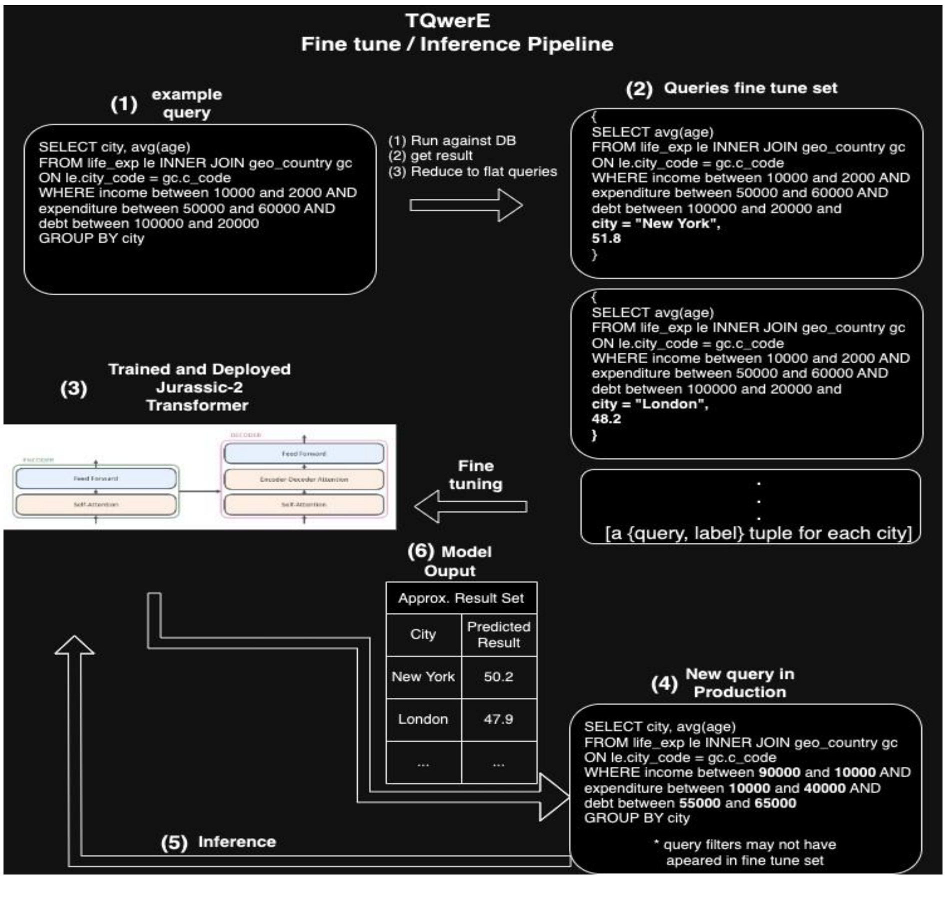
1. An AQP [29] (referred as AQP), an LSTM (long short-term memory) sequential NN, which we first trained on all datasets and then ran to extract SQL query approximations for the validation set.
2. TAPAS -A LLM developed by Google for question answering tasks over tabular data. This method was fine-tuned the same way TQwerE was trained.
3. A smart sampling method, referred as SS ([25]), which we used to derive representative data samples from the original datasets, and then we ran the testing set SQL queries against those data samples. The SS method used 30% sampling rate.

The AQP method [29] is designed and trained to limit the need to run queries against large databases, thus reducing query latency. Figures 2 and 1 respectively provide an illustration of the AQP method and our TQwerE method. We intentionally used the AQP method, as it is NN (LSTM) with a backward attention mechanism unlike Jurassic which is a type of GPT. Unlike the AQP method, a GPT has both backward and forward attention mechanisms, and it is designed to help the model focus on the most relevant parts of an input (SQL query) when making predictions. During the evaluation, our method TQwerE required less training time than the AQP method required, since the Jurassic LLM foundation model is fine-tuned as opposed to the AQP method which needs to be trained from scratch.

For example, the AQP method required a long training time (21 hours or more when trained to GTX-1080 GPU machine). For comparison, TQwerE longest fine-tuning process took 82 minutes, as TQwerE uses a pre-trained foundation LLM model and employs a short and focused fine-tuning process. Datasets. We evaluated TQwerE on nine public open-source datasets. The dataset's characteristics are presented in Table 2. This set of datasets was selected to maximize domain diversity and statistical characteristics (specifically the number of rows and target variable standard deviation (referred as STD). Dataset partitioning. Each dataset was divided into three parts:

1. a fine-tuning (training) set consisting of 70% of the queries
2. a validation set consisting of 15% of the queries, and a
3. a test set consisting of 15% of the queries.

The training set was used to train the AQP LSTM network ([29]). The finetuning set was randomly sampled (with a sampling rate of 10%) from the training set. The test set was used to evaluate the success of the entire training process, while the validation set was used to tune the learning rate and batch size hyper parameters of both the AQP and TQwerE models.
Handling overfitting. To avoid overfitting, i.e., a scenario in which the model memorizes the training set and fails to generalize on the test set, the following steps were performed:


TQwerE
Fine tune / Inference Pipeline
(1) example query
(1) Run against DB
(2) get result
(3) Reduce to flat queries
(2) Queries fine tune set
[a {query, label} tuple for each city]
(3) Trained and Deployed Jurassic-2 Transformer
Feed Forward
Self-Attention
Feed Forward
Encoder-Decoder Attention
Self-Attention
Fine tuning
(6) Model Ouput
(4) New query in Production
(5) Inference

```
SELECT city, avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 2000 AND
expenditure between 50000 and 60000 AND
debt between 100000 and 20000
GROUP BY city
```

```
{
SELECT avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 2000 AND
expenditure between 50000 and 60000 AND
debt between 100000 and 20000 and
city = "New York",
51.8
}
```

```
{
SELECT avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 2000 AND
expenditure between 50000 and 60000 AND
debt between 100000 and 20000 and
city = "London",
48.2
}
```

Approx. Result Set	
City	Predicted Result
New York	50.2
London	47.9
...	...

```
SELECT city, avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 90000 and 10000 AND
expenditure between 10000 and 40000 AND
debt between 55000 and 65000
GROUP BY city
```

* query filters may not have apeared in fine tune set

Fig.1: TQwerE method pipeline for Jurassic-2.

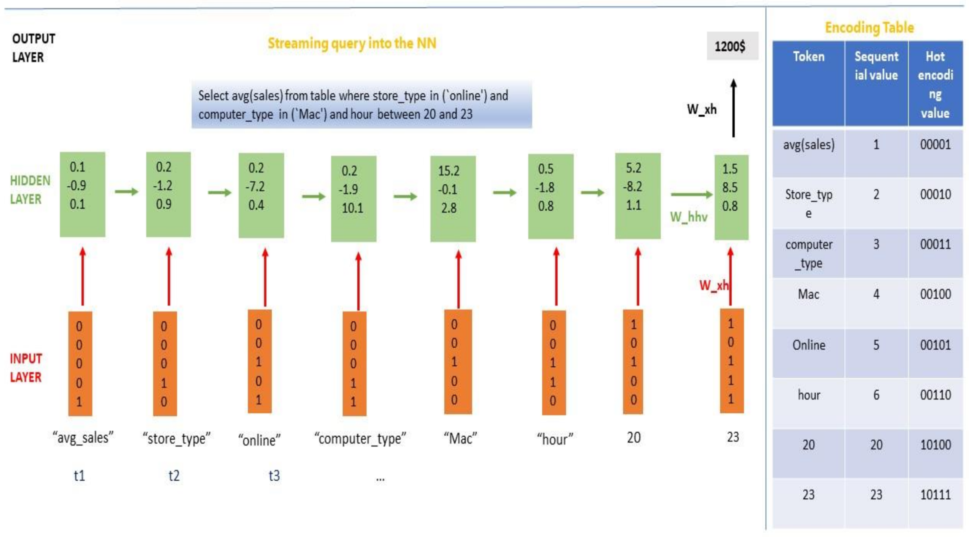
(1) All models (for each dataset) were validated during optimization (backpropagation) on a randomly selected validation set (i.e., the validation set); and

(2) After the last training epoch, we obtained the evaluation metric values for the predictions made on a second testing set (i.e., the test set) which was randomly selected before the training took place.

Steps 1 and 2 were repeated 30 times, and the average was used when evaluating our method's performance on each metric.
Evaluation metrics. We evaluated the accuracy of the models' predictions on the test data using the following metrics: (1) the Weighted MAPE (referred as WMAPE) and (2) QL - query latency - all described ahead.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100 \qquad (1)$$

where i represents a query from the test set, Y_i is the real query result, $\hat{Y}_i$ is the model's approximated query result, and n is the validation set size.



Select avg(sales) from table where store_type in ('online') and computer_type in ('Mac') and hour between 20 and 23

Token	Sequential value	Hot encoding value
avg(sales)	1	00001
Store_type	2	00010
computer_type	3	00011
Mac	4	00100
Online	5	00101
hour	6	00110
20	20	10100
23	23	10111

Fig.2: AQP method SQL encoding and train.

Table 2: Datasets' characteristics.

	Dataset	Target function	# $attr(n)$	# $attr(c)$	# rows	# queries	Mean entropy	Input tensor variance	Target column STD
1	Revenue	avg (revenue)	3	2	1000000000	5205078	6.293	0.154	40400000
2	success_rate	avg (build_time)	2	3	2333293	415791	2.264	5.421	19.196
3	product_pass	count (machine_id)	1	5	4000000000	811928	0.942	0.151	2350516
4	product_fail	count (machine_id)	1	5	95484	451173	0.942	0.153	8613
5	product_false_calls	count (machine_id)	1	5	350232	378111	0.942	0.202	215315
6	churn_customers	count (customer_id)	4	3	9263836	62092	2.782	0.13267	530
7	duration_call	sum (duration)	3	2	9349	100000	3.198	0.167	861
8	ibm_price	avg (close_price)	1	2	1048575	340489	0.343	0.125	471
9	realestate_price	avg (price)	3	2	22489348	508086	2.113	0.105	236
10	stock_price	avg (close_price)	2	1	63267	8721	5.703	0.157	118
11	paid_days	avg (actual_paid_days)	3	2	100000000	508365	0.451	0.099	25667553
12	build_duration	avg (duration)	1	3	22276094	325935	0.993	0.129	7487

training set size is referred to as '# queries'

$$\text{WMAPE} = \frac{\sum_{i=1}^{n} w_i \left| \frac{y_i - \hat{y}_i}{y_i} \right|}{\sum_{i=1}^{n} w_i} \times 100 \qquad (2)$$

where w_i is the weight for the i-th query, which is proportional to the row count of y_i.

$$\text{Mean QL} = \frac{1}{t} \sum_{i=1}^{t} \text{QL}_i \qquad (3)$$

where t is the total number of validation queries and QL_i is the Query Latency for the i-th query, measured in ms.

5 Results

Figures 4 and 5 present the WMAPE metric and QL results for the four examined methods: (1) our TQwerE method, (2) the AQP method, (3) TAPAS method and (4) the SS method. These results were obtained by applying these four methods on twelve testing sets with respect to the datasets described in Table 2.

As can be seen in figures 4 and 5, TQwerE outperformed other methods in terms of the WMAPE metric. Also, TQwerE was faster than other methods in QL metric. The difference WMAPE and QL were found to be statistically significant in a paired (over dataset) t-test for the WMAPE metric (TQwerE vs. TAPAS and SS-10) when (pvalue_TAPAS=0.0495, pvalue_SS=0.0305) and for the QL (pvalue_AQP=0.0409, pvalue_SS=0.0319).

To stress the importance of our fair comparison, we evaluated our experiments on 3 advanced models: (1)TQwerE which is a LLM based on Jurassic-2 with 178 billion trainable parameters and (2) Tapas which is also a LLM built and trained specifically to parse structured data and queries. TAPAS maintains 340 million trainable parameters and finally (3) AQP - which is a LSTM model (not LLM). However, the AQP method was build specifically for the task of approximating aggregated SQL queries. Out of these three model it is apparent that the largest LLM (in terms of number of trainable parameters) outperformed other methods.

6 Conclusions and Future Research

This paper introduces TQwerE, a novel method utilizing a state-of-the-art pretrained LLM (Jurassic) to approximate SQL query results. TQwerE outperformed the benchmark method: TAPAS, AQP and SS methods on vast majority of the datasets in terms of the WMAPE accuracy metric QL query performance metric. In addition, TQwerE requires less training time, since the Jurassic pretrained foundation model is used to approximate SQL queries after a short finetuning process. In addition, TQwerE method does not require customer SQL query encoding which is required by the AQP method. Instead, it uses the internal encoding learned within the pre-trained LLM. In future research, we plan to apply TQwerE to various constantly changing datasets with concept drifts that affect SQL query results.

Table 3: Training, model performance and hyper parameters.

	Data/Hyperparameters			Model Performance - WMAPE					Model Performance - Query Latency (ms)				
#	Dataset Name	LR	# Epochs	Jurassic	AQP	MySQL	TAPAS	SS-10	Jurassic	AQP	MySQL	TAPAS	SS-10
1	Revenue	0.03	50	1.123	1.571	NA	1.191	2.251	2.35	2.79	21.75	2.73	2.78
2	Success_rate	0.03	50	3.545	3.752	NA	4.060	5.501	2.23	2.55	22.48	2.45	2.48
3	Product_pass	0.06	60	0.069	0.071	NA	0.112	0.270	25.45	27.41	81.10	25.81	26.71
4	product_fail	0.03	50	0.506	0.515	NA	0.570	0.638	26.59	28.60	84.87	25.45	25.18

5	product_false$_calls$	0.03	50	0.112	0.125	NA	0.134	0.645	27.42	28.21	76.15	27.45	26.14
6	churn_customers	0.03	60	0.175	0.146	NA	0.179	0.372	3.14	3.55	8.15	3.61	3.68
7	duration_call	0.03	50	0.009	0.011	NA	0.101	0.513	2.45	2.84	8.17	2.87	2.46
8	ibm_price	0.06	50	0.601	0.614	NA	0.622	0.861	3.39	3.29	9.29	2.91	2.98
9	realestate_price	0.06	20	0.275	0.322	NA	0.323	0.691	2.58	2.81	7.55	2.65	2.89
10	stock_close_price	0.06	20	0.166	0.178	NA	0.197	0.291	3.57	4.03	13.98	4.15	20.19
11	paid_days	0.06	20	0.952	0.971	NA	0.997	1.011	1.96	2.05	8.95	2.03	2.09
12	build_duration	0.06	20	0.075	0.151	NA	0.197	1.091	3.78	4.01	8.19	4.08	4.15

REFERENCES

1. Sercan Ö Arik and Tomas Pfister. Tabnet: Attentive interpretable tabular learning.In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 6679–6687, 2021.
2. Dimitris Bertsimas, Kimberly Villalobos Carballo, Yu Ma, Liangyuan Na, LéonardBoussioux, Cynthia Zeng, Luis R Soenksen, and Ignacio Fuentes. Tabtext: a systematic approach to aggregate knowledge across tabular data structures. *arXiv preprint arXiv:2206.10381*, 2022.
3. Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. Deep neural networks and tabular data: A survey. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
4. Introducing ChatGPT. Introducing chatgpt, 2022.
5. Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In*Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.
6. Ganqu Cui, Shengding Hu, Ning Ding, Longtao Huang, and Zhiyuan Liu. Prototypical verbalizer for prompt-based few-shot tuning. *arXiv preprint arXiv:2203.09770*, 2022.
7. Xiang Deng, Ahmed Hassan Awadallah, Christopher Meek, Oleksandr Polozov,Huan Sun, and Matthew Richardson. Structure-grounded pretraining for text-tosql. *arXiv preprint arXiv:2010.12773*, 2020.
8. Xiang Deng, Huan Sun, Alyssa Lees, You Wu, and Cong Yu. Turl: Table understanding through representation learning. *ACM SIGMOD Record*, 51(1):33–40, 2022.
9. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pretraining of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.
10. Tuan Dinh, Yuchen Zeng, Ruisu Zhang, Ziqian Lin, Michael Gira, Shashank Rajput, Jy-yong Sohn, Dimitris Papailiopoulos, and Kangwook Lee. Lift: Languageinterfaced fine-tuning for non-language machine learning tasks. *Advances in Neural Information Processing Systems*, 35:11763–11784, 2022.
11. Julian Martin Eisenschlos, Syrine Krichene, and Thomas Müller. Understandingtables with intermediate pre-training. *arXiv preprint arXiv:2010.00571*, 2020.
12. Omer Goldman, Veronica Latcinnik, Udi Naveh, Amir Globerson, and JonathanBerant. Weakly-supervised semantic parsing with abstract examples. *arXiv preprint arXiv:1711.05240*, 2017.
13. Léo Grinsztajn, Edouard Oyallon, and Gaël Varoquaux. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, 35:507–520, 2022.
14. Asaf Harari and Gilad Katz. Few-shot tabular data enrichment using fine-tunedtransformer architectures. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1577–1591, 2022.
15. Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno,and Julian Martin Eisenschlos. Tapas: Weakly supervised table parsing via pretraining. *arXiv preprint arXiv:2004.02349*, 2020.
16. Noah Hollmann, Samuel Müller, Katharina Eggensperger, and Frank Hutter.Tabpfn: A transformer that solves small tabular classification problems in a second. *arXiv preprint arXiv:2207.01848*, 2022.
17. Arlind Kadra, Marius Lindauer, Frank Hutter, and Josif Grabocka. Well-tunedsimple nets excel on tabular datasets. *Advances in neural information processing systems*, 34:23928–23941, 2021.

18. Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, QiweiYe, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems*, 30, 2017.
19. Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo.Deep neural decision forests. In *Proceedings of the IEEE international conference on computer vision*, pages 1467–1475, 2015.
20. Roman Levin, Valeriia Cherepanova, Avi Schwarzschild, Arpit Bansal, C BayanBruss, Tom Goldstein, Andrew Gordon Wilson, and Micah Goldblum. Transfer learning with deep tabular models. *arXiv preprint arXiv:2206.15306*, 2022.
21. Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, AbdelrahmanMohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.
22. Liuqing Li, Jack Geissinger, William A Ingram, and Edward A Fox. Teachingnatural language processing through big data text summarization with problembased learning. *Data and Information Management*, 4(1):18–43, 2020.
23. Chen Liang, Mohammad Norouzi, Jonathan Berant, Quoc V Le, and Ni Lao. Memory augmented policy optimization for program synthesis and semantic parsing. *Advances in Neural Information Processing Systems*, 31, 2018.
24. Opher Lieber, Or Sharir, Barak Lenz, and Yoav Shoham. Jurassic-1: Technicaldetails and evaluation. *White Paper. AI21 Labs*, 1:9, 2021.
25. Barzan Mozafari and Ning Niu. A handbook for building an approximate queryengine. *IEEE Data Eng. Bull.*, 38(3):3–29, 2015.
26. Thomas Mueller, Francesco Piccinno, Massimo Nicosia, Peter Shaw, and YaseminAltun. Answering conversational questions on structured data without logical forms. *arXiv preprint arXiv:1908.11787*, 2019.
27. Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. Canfoundation models wrangle your data? *arXiv preprint arXiv:2205.09911*, 2022.
28. Sergei Popov, Stanislav Morozov, and Artem Babenko. Neural oblivious decisionensembles for deep learning on tabular data. *arXiv preprint arXiv:1909.06312*, 2019.
29. Nir Regev, Lior Rokach, and Asaf Shabtai. Approximating aggregated sql querieswith lstm networks. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.
30. Peng Shi, Patrick Ng, Zhiguo Wang, Henghui Zhu, Alexander Hanbo Li, Jun Wang,Cicero Nogueira dos Santos, and Bing Xiang. Learning contextual representations for semantic parsing with generation-augmented pre-training. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 13806–13814, 2021.
31. Tianze Shi, Chen Zhao, Jordan Boyd-Graber, Hal Daumé III, and Lillian Lee. Onthe potential of lexico-logical alignments for semantic parsing to sql queries. *arXiv preprint arXiv:2010.11246*, 2020.
32. Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all youneed. *Information Fusion*, 81:84–90, 2022.
33. Gowthami Somepalli, Micah Goldblum, Avi Schwarzschild, C Bayan Bruss, andTom Goldstein. Saint: Improved neural networks for tabular data via row attention and contrastive pre-training. *arXiv preprint arXiv:2106.01342*, 2021.
34. Zhiruo Wang, Haoyu Dong, Ran Jia, Jia Li, Zhiyi Fu, Shi Han, and DongmeiZhang. Tuta: Tree-based transformers for generally structured table pre-training. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 1780–1790, 2021.
35. Xiaoyu Yang, Feng Nie, Yufei Feng, Quan Liu, Zhigang Chen, and Xiaodan Zhu.Program enhanced fact verification with verbalization and graph attention network. *arXiv preprint arXiv:2010.03084*, 2020.
36. Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. Tabert: Pretraining for joint understanding of textual and tabular data. *arXiv preprint arXiv:2005.08314*, 2020.

37. Jinsung Yoon, Yao Zhang, James Jordon, and Mihaela van der Schaar. Vime: Extending the success of self-and semi-supervised learning to tabular domain. *Advances in Neural Information Processing Systems*, 33:11033–11043, 2020.

38. Tao Yu, Chien-Sheng Wu, Xi Victoria Lin, Bailin Wang, Yi Chern Tan,Xinyi Yang, Dragomir Radev, Richard Socher, and Caiming Xiong. Grappa: Grammar-augmented pre-training for table semantic parsing. *arXiv preprint arXiv:2009.13845*, 2020.

39. Min Zhang and Juntao Li. A commentary of gpt-3 in mit technology review 2021. *Fundamental Research*, 1(6):831–833, 2021.

40. Victor Zhong, Mike Lewis, Sida I Wang, and Luke Zettlemoyer. Grounded adaptation for zero-shot executable semantic parsing. *arXiv preprint arXiv:2009.07396*, 2020.

41. Wanjun Zhong, Duyu Tang, Zhangyin Feng, Nan Duan, Ming Zhou, Ming Gong,Linjun Shou, Daxin Jiang, Jiahai Wang, and Jian Yin. Logicalfactchecker: Leveraging logical operations for fact checking with graph module network. *arXiv preprint arXiv:2004.13659*, 2020.

6 AUTHORS

Nir Regev is a PhD candidate in the Dept. of Software and Information Systems Engineering at Ben-Gurion University of the Negev and an applied data scientist with 15 years of experience in various startup companies. My main research is AQP (Approximate Query Processing) with ML/DL methods.

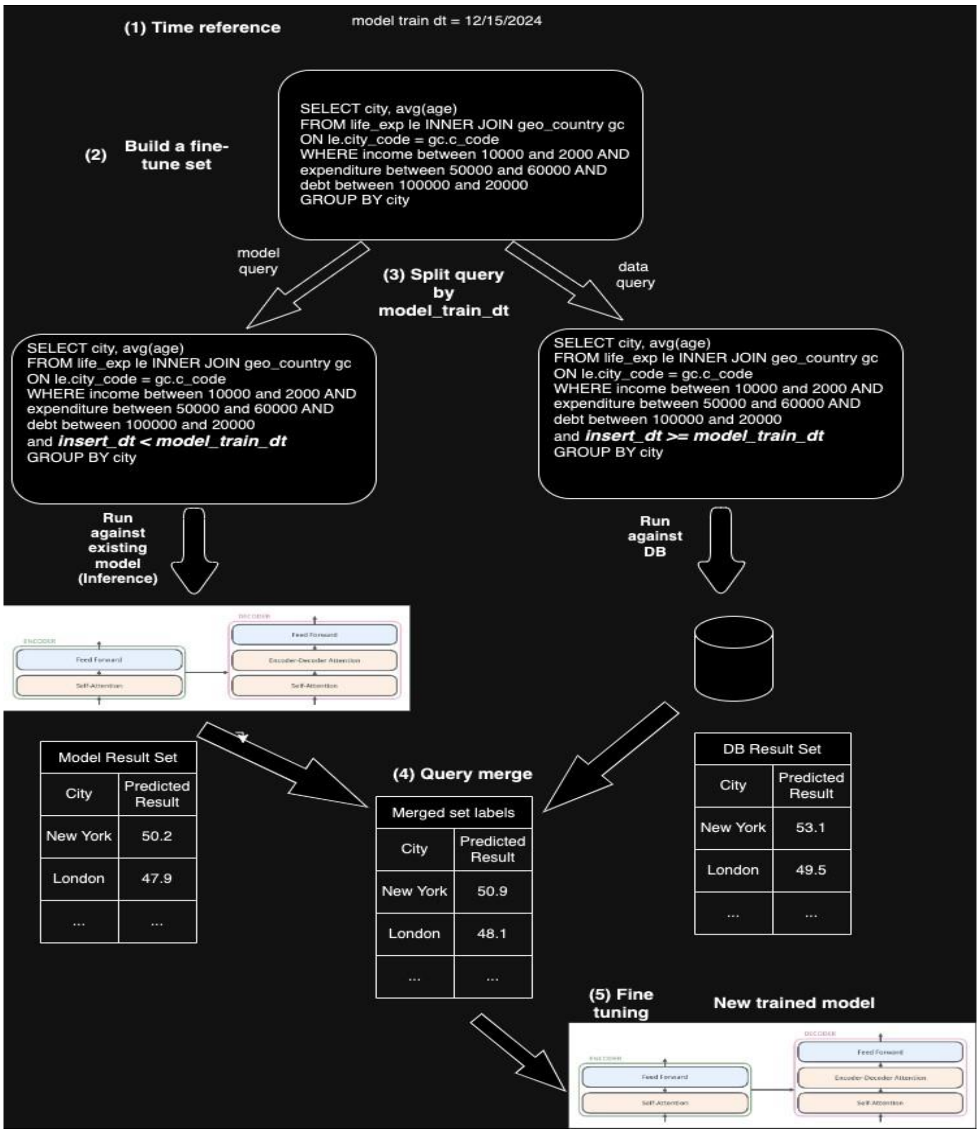
Nir Regev

Asaf Shabtai is a Professor at the Department of Information Systems Engineering at Ben-Gurion University. My main research interests are in the fields of Machine Learning, Data Mining, Temporal Data Mining Recommender Systems, Cyber Security, Fraud Detection, Customer Analytics and Social Network.

Asaf Shabtai

Lior Rokach is a data scientist and a Full Professor of Information Systems and Software Engineering. He currently serves as the chair of the Department of Software and Information System Engineering at the Ben-Gurion University of the Negev. His research interests lie in the areas of Data Science, Machine Learning, Big Data, Deep Learning and Data Mining

Lior Rokach



```
SELECT city, avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 2000 AND
expenditure between 50000 and 60000 AND
debt between 100000 and 20000
GROUP BY city
```

```
SELECT city, avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 2000 AND
expenditure between 50000 and 60000 AND
debt between 100000 and 20000
and insert_dt < model_train_dt
GROUP BY city
```

```
SELECT city, avg(age)
FROM life_exp le INNER JOIN geo_country gc
ON le.city_code = gc.c_code
WHERE income between 10000 and 2000 AND
expenditure between 50000 and 60000 AND
debt between 100000 and 20000
and insert_dt >= model_train_dt
GROUP BY city
```

Model Result Set	
City	Predicted Result
New York	50.2
London	47.9
...	...

DB Result Set	
City	Predicted Result
New York	53.1
London	49.5
...	...

Merged set labels	
City	Predicted Result
New York	50.9
London	48.1
...	...

Fig.3: TQwerE data changes adaptation method.

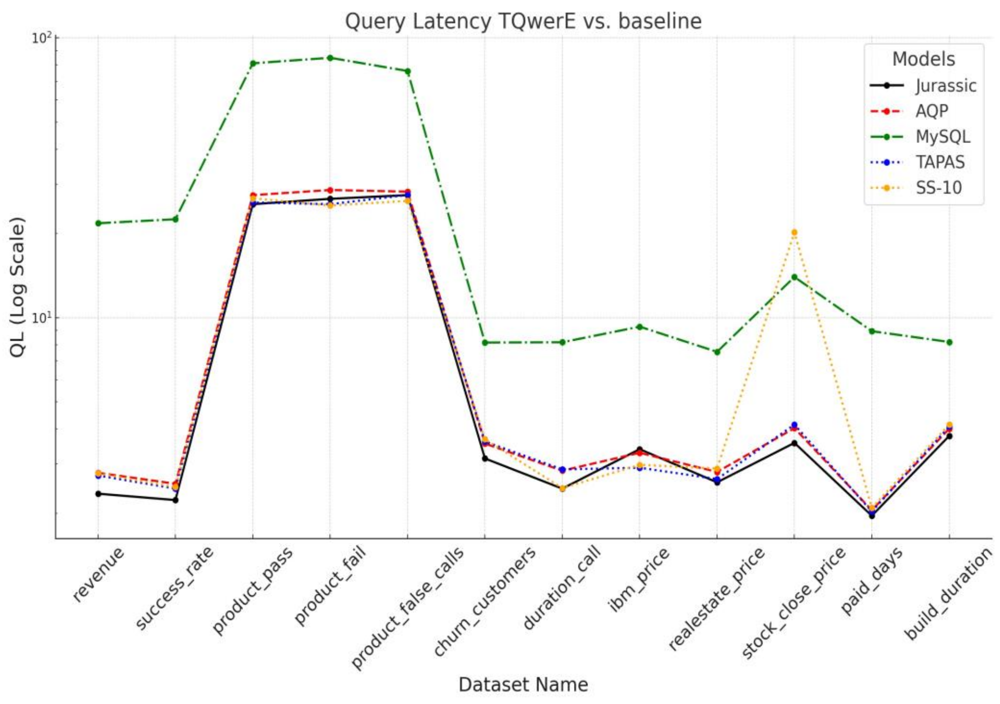


Fig.4: Query Latency TQwerE vs baseline

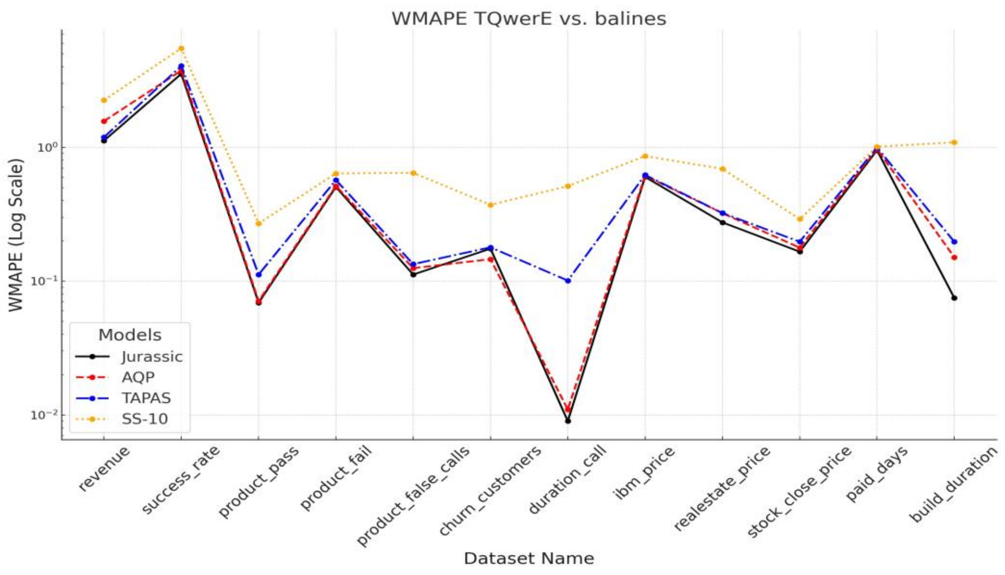


Fig.5: WMAPE comparison of AQP and Jurassic (TQwerE), TAPAS and SS.