TorchQL: A Programming Framework for Integrity Constraints in Machine Learning

AADITYA NAIK, University of Pennsylvania, USA
ADAM STEIN, University of Pennsylvania, USA
YINJUN WU, University of Pennsylvania, USA
MAYUR NAIK, University of Pennsylvania, USA
ERIC WONG, University of Pennsylvania, USA

Finding errors in machine learning applications requires a thorough exploration of their behavior over data. Existing approaches used by practitioners are often ad-hoc and lack the abstractions needed to scale this process. We present TorchQL, a programming framework to evaluate and improve the correctness of machine learning applications. TorchQL allows users to write queries to specify and check integrity constraints over machine learning models and datasets. It seamlessly integrates relational algebra with functional programming to allow for highly expressive queries using only eight intuitive operators. We evaluate TorchQL on diverse use-cases including finding critical temporal inconsistencies in objects detected across video frames in autonomous driving, finding data imputation errors in time-series medical records, finding data labeling errors in real-world images, and evaluating biases and constraining outputs of language models. Our experiments show that TorchQL enables up to 13x faster query executions than baselines like Pandas and MongoDB, and up to 40% shorter queries than native Python. We also conduct a user study and find that TorchQL is natural enough for developers familiar with Python to specify complex integrity constraints.

CCS Concepts: • **Software and its engineering → Domain specific languages**.

Additional Key Words and Phrases: Machine Learning, Integrity Constraints, Query Languages

ACM Reference Format:
Aaditya Naik, Adam Stein, Yinjun Wu, Mayur Naik, and Eric Wong. 2024. TorchQL: A Programming Framework for Integrity Constraints in Machine Learning. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 124 (April 2024), 31 pages. https://doi.org/10.1145/3649841

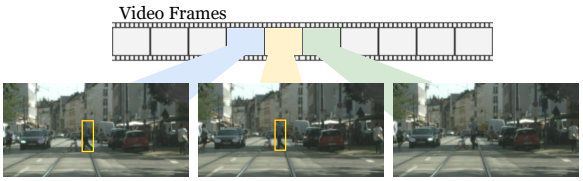
1 INTRODUCTION

Machine learning models can fail in unexpected and harmful ways. Examples include fatalities caused by self-driving vehicles [Wakabayashi 2018], vision models performing worse on people with darker skin [Wilson et al. 2019], language models producing text containing offensive stereotypes [Abid et al. 2021], and medical diagnosis models degrading in performance when used in new hospitals [Zech et al. 2018]. Identifying and avoiding such behaviors is crucial to ensuring performance, reliability, and trustworthiness of machine learning applications.

Finding and characterizing these behaviors is difficult. As a running example, consider Figure 1a, which depicts three consecutive video frames from the Cityscapes self-driving dataset [Cordts et al.

Authors' addresses: Aaditya Naik, University of Pennsylvania, Philadelphia, USA, asnaik@seas.upenn.edu; Adam Stein, University of Pennsylvania, Philadelphia, USA, steinad@seas.upenn.edu; Yinjun Wu, University of Pennsylvania, Philadelphia, USA, wuyinjun@seas.upenn.edu; Mayur Naik, University of Pennsylvania, Philadelphia, USA, mhnaik@seas.upenn.edu; Eric Wong, University of Pennsylvania, Philadelphia, USA, exwong@cis.upenn.edu.

ACM 2475-1421/2024/4-ART124
https://doi.org/10.1145/3649841



(a) Model fault visualization.

```
Query('temporal_consistency', base='preds')
  .join('preds', key=fid_plus1, fkey=fid)
  .join('preds', key=fid_plus2, fkey=fid)
  .project(match_three_bboxes)
  .project(zip_adj_preds).flatten()
  .filter(lambda sid, frames, lb, bx: (
     lb[0] == lb[1] and lb[2] == "No Match"
     and is_center(bx[0])))
```

(b) Integrity constraint as a query.

Fig. 1. (a) shows three consecutive frames in a video from the Cityscapes self-driving object-detection dataset. OneFormer, a state-of-the-art model, detects pedestrians in the center of the first two frames but not in the third. We formalize this fault as a violation of an integrity constraint. (b) shows a TorchQL query to find all such violations over a dataset named 'preds' consisting of video frames along with the model's predictions.

2016]. A state-of-the-art vision model, OneFormer [Jain et al. 2022], fails to predict a pedestrian in the third frame. This is a safety-critical issue since predicted objects should not suddenly disappear, especially not pedestrians. This model failure, however, is difficult to find since the error occurs sparsely within the 15K validation samples and it has minimal impact on most numeric evaluation metrics, so even the highest-performing models exhibit such faults.

Similar to the above example, errors often exhibit sparsely in model predictions over vast datasets. As such, in order to identify and characterize these errors, one must sift through large numbers of correct predictions. Moreover, simply identifying the errors may not be enough; oftentimes we must identify the underlying cause of these errors. In general, erroneous model predictions can be cast as violations of certain properties. For instance, the error in Figure 1a violates the property of object permanence across three adjacent frames in a video sequence.

We desire a general mechanism for specifying and checking such properties over data. *Integrity Constraints*, originally studied in the context of databases [Godfrey et al. 1998], constitute such a mechanism. As such, we can cast the problem of detecting and avoiding these errors as the problem of specifying appropriate integrity constraints and evaluating them over model predictions. For example, the error in Figure 1a can be cast as the violation of the constraint: *for every sequence of three consecutive frames, a person detected in the first two frames should be detected in the third.*

Existing machine learning frameworks lack adequate abstractions to support integrity constraints. Writing such constraints from scratch in Python typically also requires writing the infrastructure code needed to execute them. This places a burden on the user who may not know the constraint a priori and may want to test out multiple candidate constraints. Furthermore, once these constraints are specified, executing them over large scale data may require further optimizing the code, which is another burden on the user.

While frameworks such as Pandas and MongoDB can potentially abstract away some of the required infrastructure, they either have ad-hoc support for Python or come with complex interfaces and strict schema requirements. Furthermore, they may not naturally support the vast number of possible modalities that machine learning models operate over, from bounding boxes over video frames to unstructured text or audio data. This is also a challenge for systems such as LMQL [Beurer-Kellner et al. 2023] that seek to enforce integrity constraints over model outputs at runtime but are only applicable to the domain of language modeling.

We therefore observe that any effective framework for programming integrity constraints must address the following challenges:

(1) *Scalability:* it must be able to check integrity constraints in large-scale machine learning settings,
(2) *Interactivity:* it must support interactive testing and inspection of constraint violations, and

(3) *Expressivity:* it must be able to support integrity constraints specified over a diverse landscape of models, datasets, and use-cases.

In this paper, we present TorchQL, a framework that satisfies the above criteria. TorchQL enables users to specify integrity constraints as *queries*. For this purpose, it introduces a query language that seamlessly integrates relational algebra with functional programming. Figure 1b shows an example query that specifies the constraint that detects the error in Figure 1a. TorchQL executes the queries over a database representing datasets and model predictions to find violations of the corresponding integrity constraints. This allows developers to quickly and scalably test potential bugs on large datasets and models. Furthermore, the compositionality of TorchQL queries enables the rapid refinement of previous iterations to interactively prototype new potential issues and reduce false alarms among discovered faults.

TorchQL queries are easy to write yet highly expressive, capable of representing complex faults with only eight operators and simple user-defined functions. For example, the query in Figure 1b uses only four table operators—`join`, `project`, `filter` and `flatten`—along with three user-defined functions to search for violations of the object permanence constraint. This query finds only 627 frames containing such errors out of 15,000, allowing it to effectively detect errors in large-scale machine learning tasks, while systems like Python require considerable optimizations to do so without timing out. We discuss this example in more detail in Section 6.1.

We demonstrate how TorchQL enables the use of integrity constraints for detecting and fixing correctness problems in a diverse set of machine learning tasks—object detection, data imputation, image classification, text generation, and natural language reasoning—across domains involving self-driving videos, time-series medical records, and large language models (LLMs). We show the effectiveness of TorchQL on these tasks and domains through five extensive case studies. Moreover, our evaluation shows that TorchQL enables faster query executions (up to 13x best-case speed-ups) than baseline systems such as Pandas and MongoDB, and more concise queries (up to 40% shorter) than native Python. We also conduct a user study with 10 users to evaluate the usability of TorchQL. We find that they were able to quickly learn TorchQL to program integrity constraints and find model mispredictions.

We summarize the contributions of the paper:

(1) We propose integrity constraints as a means for discovering and characterizing faults in general machine learning tasks.
(2) We develop TorchQL[1], a framework for programming and checking integrity constraints over models and datasets in a manner that is *scalable*, *interactive*, and *expressive*. At its core, TorchQL provides abstractions to specify and evaluate integrity constraints as database queries.
(3) We perform extensive experiments that demonstrate the efficiency and conciseness of TorchQL for integrity constraint queries in a multitude of use-cases across a variety of domains.
(4) We also complement our experiments with a user study and multiple case studies to validate the usability and expressiveness of TorchQL.

2 ILLUSTRATIVE OVERVIEW

In this section, we illustrate how TorchQL is used to iteratively discover and check integrity constraints to detect model faults, and contrast with programming the same constraints in native Python. This process is done offline, once the model is trained, but before it is deployed. In Section 6, we discuss other use-cases, such as using the constraints to detect model faults at runtime.

[1]The TorchQL system is currently available at https://github.com/TorchQL/torchql/.

```
Query('people_count', base='preds')
  .filter(lambda p:
    count_people(p['labels']) != 0)
```

(a) TorchQL

```
people_count = []
for p in preds:
  if count_people(p['labels']) != 0:
    people_count.append(p)
```

(b) Native Python

Fig. 3. Queries for finding individual frames in which at least one pedestrian is predicted.

```
Query('changing_people', base='preds')
  .join('preds', key=fid_plus_1, fkey=fid)
  .filter(lambda f1, f2:
    count_people(f1['labels']) !=
    count_people(f2['labels']))
```

(a) TorchQL

```
changing_people = []
for p in preds:
  for q in preds:
    if fid_plus_1(p) == fid(q):
      if count_people(p['labels']) !=
         count_people(q['labels']):
        changing_people.append([p, q])
```

(b) Native Python

Fig. 4. Queries for finding pairs of consecutive frames with different predicted pedestrian counts.

Consider the error in Figure 1 where a self-driving object detector fails to detect a pedestrian in the third frame of a video sequence. Since this is a critical bug, a machine learning practitioner would want to find other instances of this error, both within the training data as well as when the model is deployed. We therefore seek to *characterize* this error as an integrity constraint violation. Before we can accomplish this, however, we must first discover the integrity constraint itself.

We first initialize a TorchQL database `db` as shown in Figure 2 and populate it with a table named `'preds'` containing predictions of OneFormer over 15,000 frames. Any query `q` we write can then be evaluated over this table by invoking it over `db`.

```
preds = ...  # frames and predictions
db = Database()
db.register(preds, 'preds')
q = Query(...)  # creating a query
result = q(db)  # running the query
```

Fig. 2. Initializing a TorchQL database and running queries over it.

Since we do not know the constraint beforehand, we start with a simple initial constraint: frames in which at least one pedestrian is predicted. This can be written as a simple TorchQL query as shown in Figure 3a. Here, we specify the query name (`people_count`) and the table over which the query operates (`preds`). Each prediction consists of the bounding boxes and their corresponding labels for each frame. We supply a *user-defined function* to the `filter` operator, which executes it over each frame when the query is run, and keep the predictions that satisfy the condition.

We also show the same constraint implemented in native Python in Figure 3b. The lack of query abstractions in Python means that we need to write a `for` loop, an `if` condition, and manually add satisfying predictions to the data structure (`people_count`). As such, the lack of query abstractions often necessitates one to implement both *what* the constraint is, as well as *how* to evaluate it.

Evaluating this constraint gives us 4,591 frames out of the total 15,000 frames, many of them false positives. This means that the constraint is too broad and must be refined further. A subsequent hypothesis may look for two consecutive predictions which differ in the number of people detected. In such a case, it is possible that some people predicted in the first frame are missed in the second.

We can construct the query in Figure 4a to represent this hypothesis by chaining more table operators to the original query. First, we use a `join` operator to join the table of pedestrians with the same table shifted forward in time by one frame to get a new table containing pairs of consecutive predictions. Here too, we use user-defined functions to specify the key and foreign key of the join. This gives us a table of consecutive predictions. Observe that TorchQL allows its queries to directly

```
Query('temporal_consistency', base='preds')
  .join('preds', key=fid_plus_1, fkey=fid)
  .join('preds', key=fid_plus_2, fkey=fid)
  .project(match_three_bboxes)
  .project(zip_adj_preds)
  .flatten()
  .filter(lambda sid, frames, lb, bx:
    lb[0] == lb[1] and
    lb[2] == "No Match" and
    is_center(bx[0]))
```

(a) TorchQL

```
temporal_consistency = []
for p in preds:
  for q in preds:
    for r in preds:
      if fid_plus_1(p) == fid(q) and
         fid_plus_2(p) == fid(r):
        bboxes = match_three_bboxes(p, q, r)
        bboxes = zip_adj_preds(*bboxes)
        for box in bboxes:
          for (sid, frames, lb, bx) in box:
            if lb[0] == lb[1] and
               lb[2] == "No Match" and
               is_center(bx[0]):
              temporal_consistency.append(box)
```

(b) Native Python

Fig. 5. Queries for finding sequences of three consecutive frames where a pedestrian is detected in the center of the first two frames but not in the third.

run on the objects being queried themselves. This allows us to seamlessly integrate relational algebra with lambda functions, as queries can contain and execute arbitrary Python functions (e.g., `fid` and `fid_plus_1`) over these objects without needing to define an explicit schema.

We can then apply another `filter` operator to select only pairs of predictions with a differing number of pedestrians. This now reduces the number of filtered frames to 3,638 from the 4,591 that were filtered out in the initial query. As was the case with the previous iteration, however, the Python implementation (Figure 4b) requires writing multiple looping and conditional statements to mimic the `join` and `filter` operators.

Our latest query produced a table of consecutive predictions with a differing number of people. However, such predictions also include people that did not go missing, such as those exiting or entering the second frame. We therefore need to identify the exact persons that have eluded the object detector. This requires converting our latest table of consecutive predictions into a table of consecutive persons. We also wish to do so over three frames instead of two to avoid even more false positives, and restrict our constraint to predictions in the center of the frame to avoid issues with people leaving or entering the frame.

The TorchQL query identifying all such sequences of three consecutive frames is shown in Figure 5a. Here, the sequences are such that the pedestrian is detected in the center of the first frame, detected again in the second frame, but not in the third.

Our first step is to join an additional 'preds' table to get a set of three consecutive predictions instead of two. We again use the `join` operator, similar to how we used it in the previous query. Although we have a set of three consecutive predictions, each prediction contains an unordered list of detected objects. As a result, it is not immediately obvious which bounding boxes refer to the same person. We therefore use a well-developed tool from object detection for aligning the bounding boxes between frames [Bolya et al. 2020]. We can apply this complex function, denoted as `match_three_bboxes` to all of our prediction sequences using the `project` table operator, which applies a function to every row in a table to create a new table.

We further use a combination of other user-defined functions, table operators, and a final filter to extract consecutive objects where the object is detected as a person in the first two frames but not in the third. This final constraint refines our search of critical errors from 3638 to 627 frames out of 15,000. In contrast, the Python implementation shown in Figure 5b requires the additional join to be explicitly set up as a nested loop, and more loops to mimic the `flatten` operator. Moreover, the Python implementation now fails to scale to the 15,000 frames, timing out after 10 minutes,

Table 1. Comparison of querying systems.

Querying Systems	Data Representation	Flexible Schema	User-Defined Functions	Querying Abstractions
Pandas	`numpy.array` tables	✗	✓	✓
MongoDB [Banker et al. 2016]	key-value stores	✓	✗	✓
TorchQL	object collections	✓	✓	✓
Python, OMG [Kang et al. 2018]	arbitrary objects	✓	✓	✗

while the TorchQL query executes in around 43 seconds. While the Python implementation can be optimized, doing so adds 24 more lines of code, imposing additional user burden.

We thus show that throughout this process, TorchQL allowed us to quickly and easily execute and iterate over prototype queries on not only the original dataset but also on intermediate tables produced while trying to program the constraint to detect temporal inconsistencies. On the other hand, iterating over constraints in Python is accompanied by setting up the infrastructure to scalably execute the constraint, an issue prevalent in machine learning codebases such as the ones mentioned in Appendix B.1. Our final query involves the composition of 4 unique table operators, each of which is parameterized by a custom lambda function to handle various data types. This process is representative of how the scalability, interactivity, and expressivity of TorchQL allows us to program integrity constraints to discover errors in machine learning applications.

3 THE TORCHQL FRAMEWORK

This section presents the TorchQL framework. We first describe the underlying data model that queries operate over and then present the syntax and semantics of queries.

3.1 Data Model

While integrity constraints can be programmed using existing systems like Pandas or native Python, as we illustrated in Section 2, they are ill-equipped for our needs. If arbitrary Python objects are needed for a debugging task, Python is preferable to Pandas, since the latter does not support querying over arbitrary objects. Conversely, if the objects adhere to a rigid structure or can be easily flattened into a table of primitive datatypes, it is more efficient to take advantage of Pandas querying abstractions. Other candidates include NoSQL database systems like MongoDB and its in-memory counterpart, ArangoDB. However, they are designed to store and query over key-value pairs and do not support user-defined functions (UDFs) that allow users to leverage external modules and express arbitrarily complex queries. We summarize these tradeoffs in Table 1.

In order to be usable, TorchQL must support queries over arbitrary Python objects without requiring much data wrangling. TorchQL therefore uses an object-relational representation to represent the data being queried. It inherits Python's data model and considers an *object* (o) to be a fundamental abstraction of data. As such, each object may either be a Python primitive or instantiation of a Python class, a list or set of other objects, or a collection of key-value pairs.

These objects may be further organized into lists with other similar objects, referred to as *tables* (T). Objects in the same table are oftentimes related with each other, but need not conform to any schema, like frames from a video, or prompts for a large language model. Tables can be further assigned names and organized into a collection called a *database* (D). We show the semantic domains for objects, tables, and databases used by TorchQL in Figure 6a.

An object-relational representation has several advantages over other common representations. First, while it requires objects being queried to be contained within a collection, it has no restrictions

$$
\begin{array}{rrcl}
\text{(object)} & o & ::= & p \\
 & & | & [o_1, \ldots, o_n] \\
 & & | & \{o_1, \ldots, o_n\} \\
 & & | & \{p_1 : o_1, \ldots, p_n : o_n\} \\
\text{(table)} & T & ::= & [o_1, \ldots, o_n] \\
\text{(database)} & D & ::= & [n_1 \to T_1, \ldots, n_k \to T_k]
\end{array}
$$

(a) Semantic Domains.

$$
\begin{array}{rrcl}
\text{(name)} & n & & \\
\text{(function)} & f & & \\
\text{(query)} & Q & ::= & n \mid Q.a \\
\text{(statement)} & S & ::= & \texttt{register}(n, T) \\
 & & | & n \leftarrow Q \\
\text{(program)} & P & ::= & \epsilon \mid S; P
\end{array}
$$

$$
\begin{array}{rrcl}
\text{(operator)} & a & ::= & \texttt{join}\,(n, f_1, f_2) \\
 & & | & \texttt{filter}\,(f) \mid \texttt{flatten}\,() \\
 & & | & \texttt{project}\,(f) \\
 & & | & \texttt{order\_by}\,(f) \\
 & & | & \texttt{group\_by}\,(f) \\
 & & | & \texttt{unique}\,() \mid \texttt{reduce}\,(f)
\end{array}
$$

(b) Abstract syntax.

Fig. 6. Core language of TorchQL.

on the structure of the objects being queried. This allows for the flexibility to query over models and datasets without the associated overhead of converting the data into a compatible representation.

Moreover, storing these objects within collections allows for enough structure within each table to provide succinct yet powerful querying abstractions over them akin to their relational algebra counterparts. Operations such as joining tables or grouping rows turn into single-line specifications without the need for setting up low-level infrastructure that would otherwise be required.

An object-relational representation also allows for the support of executing UDFs over the objects within each table. TorchQL supports this by allowing users to supplement each relational algebra abstraction with one or more UDFs if needed. This also allows the user to leverage external libraries, machine learning models, and other tools to write their queries within the TorchQL framework.

3.2 The Query Language

We now describe TorchQL's query language. We first present its syntax, shown in Figure 6b, followed by the operational semantics, shown in Figure 7.

3.2.1 Syntax. A TorchQL *program* comprises a sequence of statements each of which defines a named base table or a named query. Each query is a chain of table operators that define a sequence of transformations to apply to one or more tables in the database. TorchQL provides eight unique table operators. These operators, with the exception of `flatten` and `unique`, are empowered by user-defined functions (UDFs). UDFs are arbitrary Python functions that enable powerful transformations over each object of a given table, or in the case of `reduce`, over the entire table. Note that since UDFs are arbitrary functions, they may result in potentially non-terminating queries depending on their definitions. We discuss these operators in more detail in Section 3.2.3, after describing the program semantics.

3.2.2 Semantics. A TorchQL program is interpreted as a sequence of database transformations induced by its constituent statements. Statement $\texttt{register}(n, T)$ adds base table T with the name n to the database whereas statement $n \leftarrow Q$ runs query Q over tables in the database to result in a new table which is added to the database with the name n. In particular, Q is comprised of the name n of an existing table and a chain of zero or more table operators $a_1.a_2 \ldots a_k$. When executing Q over database D, table $T_0 = D[n]$ is first retrieved and then the table operations specified in Q sequentially transform the table. Starting from table T_0, operation a_i transforms table T_{i-1} into T_i to eventually produce the output table T_k.

These semantics are vital for TorchQL to be effective at interactively programming integrity constraints. Iterating over previous hypotheses requires the results of executed queries to be stored and usable across programming sessions. Moreover, since the TorchQL language allows a single query to be arbitrarily complex, these semantics allow for simpler queries by decomposing complex operations into multiple queries. Queries can also act as preprocessors, allowing for features to be extracted from unstructured data before using those features to program integrity constraints. Sequentially executing and storing results of previous statements allows for these use cases.

Program semantics

$$\frac{}{D \vdash \epsilon \triangleright D}\ [\textsc{Program\_E}] \qquad \frac{D \vdash Q \triangleright T \qquad D, T \vdash a \triangleright T'}{D \vdash Q.a \triangleright T'}\ [\textsc{Query\_Op}] \qquad \frac{}{D \vdash n \triangleright D[n]}\ [\textsc{Query\_N}]$$

$$\frac{D[n \mapsto T] \vdash P \triangleright D'}{D \vdash \texttt{register}(n, T); P \triangleright D'}\ [\textsc{Program\_Reg}] \qquad \frac{D \vdash Q \triangleright T \qquad D[n \mapsto T] \vdash P \triangleright D'}{D \vdash n \leftarrow Q; P \triangleright D'}\ [\textsc{Program\_Query}]$$

Operator semantics

$$\begin{array}{cccccccccc} f & : & \mathbb{O} \to \mathbb{O}, & & s & : & \mathbb{O} \to \textsc{Bool}, & & g & : & \mathbb{T} \to \mathbb{T}, \\ \sigma_s & : & \mathbb{T} \to \mathbb{T}, & \cdot \bowtie_{f_K, f_{FK}} \cdot & : & \mathbb{T} \to \mathbb{T}, & & \cdot +\!\!+ \cdot & : & \mathbb{T} \to \mathbb{T} \end{array}$$

$$\frac{}{D, T \vdash \texttt{join}(n, f_K, f_{FK}) \triangleright T \bowtie_{f_K, f_{FK}} D[n]}\ [\textsc{Join}]$$

$$\frac{D, T \vdash \texttt{project}(f) \triangleright T'}{D, t : T \vdash \texttt{project}(f) \triangleright f(t) : T'}\ [\textsc{Project1}] \qquad \frac{}{D, [\,] \vdash \texttt{project}(f) \triangleright [\,]}\ [\textsc{Project2}]$$

$$\frac{}{D, T \vdash \texttt{reduce}(g) \triangleright g(T)}\ [\textsc{Reduce}] \qquad \frac{}{D, T \vdash \texttt{filter}(s) \triangleright \sigma_s T}\ [\textsc{Filter}]$$

$$\frac{\begin{array}{c} D, T \vdash \texttt{filter}(\lambda x.f(x) \leq f(t)) \triangleright T_{\leq x} \\ D, T \vdash \texttt{filter}(\lambda x.f(x) > f(t)) \triangleright T_{>x} \\ D, T_{\leq x} \vdash \texttt{order\_by}(f) \triangleright L \\ D, T_{>x} \vdash \texttt{order\_by}(f) \triangleright R \end{array}}{D, t : T \vdash \texttt{order\_by}(f) \triangleright L +\!\!+ [t] +\!\!+ R}\ [\textsc{Order1}] \qquad \frac{\begin{array}{c} D, T \vdash \texttt{filter}(\lambda x.f(x) = f(t)) \triangleright T_= \\ D, T \vdash \texttt{filter}(\lambda x.f(x) \neq f(t)) \triangleright T_{\neq} \\ D, T_{\neq} \vdash \texttt{group\_by}(f) \triangleright T' \end{array}}{D, t : T \vdash \texttt{group\_by}(f) \triangleright (f(t), t : T_=) : T'}\ [\textsc{Group1}]$$

$$\frac{}{D, [\,] \vdash \texttt{order\_by}(f) \triangleright [\,]}\ [\textsc{Order2}\ ] \qquad \frac{}{D, [\,] \vdash \texttt{group\_by}(f) \triangleright [\,]}\ [\textsc{Group2}]$$

$$\frac{D, T \vdash \texttt{filter}(\lambda x.x \neq t) \triangleright T_{\neq} \quad T_{\neq}, D \vdash \texttt{unique}() \triangleright T'}{D, t : T \vdash \texttt{unique}() \triangleright t : T'}\ [\textsc{Unique1}] \qquad \frac{}{D, [\,] \vdash \texttt{unique}() \triangleright [\,]}\ [\textsc{Unique2}]$$

$$\frac{D, T \vdash \texttt{flatten}() \triangleright T'}{D, t : T \vdash \texttt{flatten}() \triangleright [t] +\!\!+ T'}\ [\textsc{Flatten1}] \qquad \frac{}{D, [\,] \vdash \texttt{flatten}() \triangleright [\,]}\ [\textsc{Flatten2}]$$

Fig. 7. Operational semantics of TorchQL.

3.2.3 Operators. We now describe the operator semantics of TorchQL. The eight table operators are largely drawn from relational algebra and natively support complex operations over tables. In general, each operator takes as input a table, and produces a table as output. With the exception of `flatten` and `unique`, users can supplement these operators with UDFs. However, the operators differ in how these supplied UDFs are executed over the objects within each table.

(1) *Join.* This operator composes objects from two tables into a single table. The composition is achieved through the supplied UDFs $f_K : \mathbb{O} \to \mathbb{O}$ and $f_{FK} : \mathbb{O} \to \mathbb{O}$, as shown in Join in Figure 7:

$$T_i \bowtie_{f_K, f_{FK}} T_j = [\,[o_i, o_j] | o_i \in T_i, o_j \in T_j, f_K(o_i) = f_{FK}(o_j)\,]$$

Here, $\mathbb{O}$ denotes the domain of all objects. Since the key-foreign key pairs for the tables are defined by the results of UDFs, one can join tables based on values that can be results of arbitrarily complex operations not existing within the tables. An example of this is in the `join` of the query in Figure 1, where the function `fid_plus_1` returns the frame ID of the next frame, while `fid` returns the frame ID of the current frame. Using these functions allows one to join the table containing all the frames with itself to produce a table containing pairs of consecutive frames.

This join is a form of equijoin, and can thus be implemented as a *hash join*, where we use a hash table to join the objects from table T_i to those of T_j. Using a hash join algorithm allows us to achieve this with a complexity of $O(m + n)$, where $m = |T_i|$ and $n = |T_j|$, as opposed to typical join algorithms that perform with complexity $O(mn)$.

(2) *Project.* This operator transforms each object in a table according to the supplied UDF $f : \mathbb{O} \to \mathbb{O}$ as described in Project1 and Project2 in Figure 7. This allows us to perform powerful and flexible transformations on the rows of the table, such as including image transformations using image processing libraries, or analyzing the inferences made by external models as shown in the query in Figure 10b where the `project` function is used to generate prompts for the LLMs.

(3) *Filter.* This operator uses the supplied UDF $s : \mathbb{O} \to$ Bool to filter out rows from a table. This is depicted in Filter in Figure 7 using the selection operator σ from relational algebra.

(4) *Order By.* This operator reorders the objects of the table T according to the supplied UDF $f : \mathbb{O} \to \mathbb{O}$ as shown in Order1 and Order2 in Figure 7. Again, the UDF allows the reordering of objects without the need to explicitly populate the table with the values to order the objects by.

(5) *Group By.* This operator allows grouping objects of a table into subtables, where each subtable contains the objects that produce the same value when passed to the UDF $f : \mathbb{O} \to \mathbb{O}$. This operator is unique in the sense that its result contains nested tables that can be further manipulated using table operators. This allows us to perform powerful group-by operations on the table, such as grouping by the hour of the day or the sequence ID as shown in the query in Figure 14a.

(6) *Flatten.* This operator flattens each object $o_j \in T$ where o_j is a collection $[o_{j1}, o_{j2}, o_{j3}, \ldots]$ to produce table T' such that each element $o_{jm} \in T'$. If o_j is not a collection, then it is left unchanged.

(7) *Unique.* This operator returns a table T' where duplicate rows from T are removed.

(8) *Reduce.* This operator is different from the previously discussed operators in that it runs the supplied UDF $g : \mathbb{T} \to \mathbb{T}$ over the *entire table* rather than over individual objects within that table. Here, $\mathbb{T}$ is the domain of tables. This not only allows for general aggregation functions like count, length, min, max, and others, but also for recursive functions over tables to be run as a part of a TorchQL query without having to switch to native Python.

The use of UDFs within these operators enables a versatile querying system. First, since the UDFs can be specified directly over objects, users do not need to wrangle the data to fit a schema in order to efficiently query a model's predictions. Second, unlike in other querying systems, extracting features necessary for building these queries now becomes integrated into the query writing process, rather than being a separate component that practitioners independently refine. Finally, since UDFs can be arbitrary Python code, TorchQL is Turing complete.

4 IMPLEMENTATION

TorchQL has been implemented in Python which is the primary language used for machine learning pipelines. The underlying database system is designed to be in-memory. This design choice eliminates the overhead of storing and retrieving objects from disk while executing queries, thereby reducing the turnaround time for each query. As such, all tables being queried over are permanently loaded in the main memory for as long as the programming session lasts. While the data for the tables may be stored to disk and retrieved, they need to be loaded into memory before writing queries over them.

When writing queries, the tables that they access must exist in the database before they can be executed. Therefore, when a database is newly initialized, as is the case when starting a programming session for a new task, it must be populated with these tables using the `register` function. This is typically used for loading in the training, validation, or test datasets, after which model predictions can be obtained over them via TorchQL queries.

The TorchQL database engine compiles each TorchQL query, comprising a pipeline of operators, into an executable sequence of table operations. The engine applies relevant optimizations while compiling the query. TorchQL relies on hash data structures to perform most of these optimizations,

though batch-driven optimizations are also discussed later. One example where hashing allows for TorchQL to optimize queries is in the `join` operation. Since TorchQL joins are analogous to SQL equijoins, they are implemented as hash joins, where the results of the key and foreign key functions are hashed. This results in a complexity of $O(M + N)$, where M and N are the number of rows in the tables being joined, as opposed to $O(MN)$ in the naive implementation. Hashing similarly allows TorchQL to optimize the grouping and unique operations.

In order to seamlessly integrate the data-loading process, as well as the interaction between TorchQL queries and machine learning models, we design TorchQL to support conventional machine learning frameworks, specifically PyTorch. To do so, TorchQL tables inherit the base PyTorch `Dataset` class. In other words, TorchQL tables are instantiations of PyTorch `Datasets`. This has a few advantages while working with PyTorch machine learning pipelines.

For one, users can directly register PyTorch datasets into TorchQL databases without needing to cast them as TorchQL tables explicitly, allowing them to directly query their data. For instance, we can download the training data for the MNIST dataset using PyTorch's API and directly load it into a TorchQL database as shown in Figure 8.

```
train_data = datasets.MNIST(
  root = 'data', train = True,
  transform = ToTensor(),
  download = True,
)
db = Database("mnist")
db.register(train_data, "train")
```

Fig. 8. Loading a PyTorch dataset.

This also allows TorchQL's database engine to perform batch-driven optimizations by leveraging PyTorch's Dataloader to efficiently iterate over batches of objects, as well as PyTorch's support for batch processing and vectorized tensor operations. This results in optimizing the execution of queries whose UDFs work with batched tensors.

5 EVALUATION SETUP

We now discuss the setup for evaluating the effectiveness of TorchQL as an integrity constraint programming framework. As discussed in Section 1, we require TorchQL to be scalable, interactive, and expressive. We therefore evaluate it by answering the following research questions:

RQ1. Expressivity: Can TorchQL be used to program integrity constraints in diverse settings?

RQ2. Performance: Are TorchQL queries concise and efficient on large-scale data?

RQ3. Usability: Is TorchQL intuitive and easy to use for users unfamiliar with the system?

For **RQ1**, we write queries for five machine learning tasks over domains such as self-driving videos, time-series healthcare data, trap-camera images, and natural language text. Out of 182 queries that were written, we present and analyze 11 queries in-depth across five case studies in Section 6. The tasks are summarized in Table 2 and the chosen queries are described in Table 3.

For **RQ2**, we implement seven of these queries in three baseline systems Python, Pandas, and ArangoDB to compare their conciseness and efficiency over their corresponding datasets and models. We present our findings in Section 7. For **RQ3**, we conduct a user study with 10 participants writing three queries of increasing complexity in TorchQL. We present the results in Section 8.

In the rest of this section, we describe each machine learning task from Table 3 including the chosen datasets and models, the correctness problems of interest, and the integrity constraints and queries to address them. We then proceed to the sections that answer each research question.

5.1 Object Detection

Overview and Setup. In this task, the goal is to predict bounding boxes and their labels for individual frames in self-driving videos. We consider the validation set of the Cityscapes dataset, which contains 15,000 video frames, and use the OneFormer [Jain et al. 2022] model for predicting bounding boxes and their labels for each video frame.

Table 2. Summary of the datasets and models used in the experiments.

Task	Dataset	Dataset Size	Model	Application Domain
Object Detection	Cityscapes validation set [Cordts et al. 2016]	15K	OneFormer [Jain et al. 2022]	Self-Driving Videos
Data Imputation	Physionet-2012 Challenge [Silva et al. 2012]	4000	SAITS [Du et al. 2023]	Time-Series Healthcare Data
Image Classification	iWildCam training set [Beery et al. 2020]	121K	ResNet50 [He et al. 2016]	Trap-Camera Images
Text Generation	Alpaca [Taori et al. 2023]	52K	T5 [Raffel et al. 2020]	Natural Language
Natural Language Reasoning	GSM8K [Cobbe et al. 2021] Date Understanding [et al. 2023]	1319 369	Mistral 7B [Jiang et al. 2023]	Natural Language

Correctness Problems. Object detection models, in general, are evaluated using metrics like mean average precision (mAP). This metric considers a prediction correct if it has an Intersection over Union (IoU) ratio of at least 0.5. It then aggregates the precision of all predictions over all the frames. However, in cases where critical errors are rare and sparsely distributed over the predictions, this may cause the accurate predictions to overshadow the severe errors.

Integrity Constraints. Given this issue with the mAP metric, we seek to discover critical model faults by programming integrity constraints like the one shown in Figure 1b. We wrote a total of 57 queries to investigate various potential integrity constraints and analyze two such constraints in Section 6.1. The first, denoted S-Q1 in Table 2, discovers all objects from all the 15K frames that occur in the center of one frame, somewhere in the next consecutive frame, and are not detected in the third consecutive frame. A similar query was also investigated by Kang et al. [2018]. The second, S-Q2, aims to find vehicles with extremely high speeds across three frames. These objects with outlier speeds are likely to indicate incorrect bounding box predictions since most objects (including moving cars) have limited movement between frames.

5.2 Data Imputation

Overview and Setup. In this task, the goal is to predict missing values within time-series healthcare data. We consider the Physionet-2012 Challenge's [Silva et al. 2012] medical time-series dataset. Each sample in this dataset is composed of multiple univariate time series, each of which includes records of one feature across multiple time stamps, as illustrated in Figure 9. The dataset contains 4000 time-series samples and each sample consists of 35 lab values (features), collected hourly, within a 48-hour time window. Overall, up to 80% of the values in the samples are missing. We use the state-of-the-art model SAITS [Du et al. 2023] to impute these missing values.

Correctness Problems. Missing data in healthcare settings is prevalent due to irregular patient visits or clinical errors [Lee et al. 2017]. Imputing this data is necessary to train models for downstream use-cases. As a result, it is important that the imputed data is as accurate as possible. As discussed in [You et al. 2018], medical data such as the sample in Figure 9 should respect domain knowledge, such as the Glasgow Coma Scale's range of 1 to 15, as well as common sense, such as the assumption of smooth variations in univariate time series data (henceforth called the *smoothness assumption*). We therefore aim to find imputed values that do not respect these assumptions. For instance, Figure 9

Table 3. Queries written for evaluating TorchQL. Queries with an asterisk are only used in case studies.

Task	Query	Query Description	Query Operators						
			join	*project*	*filter*	*group*	*order*	*flatten*	*UDFs*
Object Detection	S-Q1	Retrieve all sequences of three continuous frames from the Cityscapes dataset in which an object appearing in the first two frames, and the center of the first, is not detected in the third frame by the OneFormer model.	2	2	1	0	0	1	3
	S-Q2	Retrieve all vehicles predicted in three consecutive frames of the Cityscapes dataset by the OneFormer model and compute their speed across the frames.	1	1	1	0	0	0	2
Data Imputation	T-Q1	Compute the difference between all non-missing pairs of temporally consecutive feature values and compute the 99th percentile difference by following [Naik et al. 2023], for each feature.	1	2	0	0	0	0	2
	T-Q2	Compute the interquartile range of non-missing ground-truth values across time for each feature and use it to detect outliers in the SAITS model's imputations.	1	2	0	0	0	1	1
	T-Q3	Compute the interquartile range of all values (including imputed ones) across time for each feature and use it to detect outliers in the SAITS model's imputations.	0	1	0	0	0	0	1
Image Classification	I-Q1	Retrieve trap camera video sequences containing frames with more than one unique ground-truth animal.	0	1	1	1	1	0	0
	I-Q2*	Find empty trap camera frames where a ResNet model predicts an animal.	0	1	0	1	0	1	1
Text Generation	B-Q1	Retrieve adjectives used to describe the nouns 'farmer' and 'engineer' in the T5 Alpaca dataset.	0	2	0	1	0	0	2
	B-Q2*	From sets of adjectives biased towards farmers and engineers, find the adjectives which GPT-3.5 consistently chooses to describe farmers or engineers.	1	5	0	0	0	2	3
Natural Language Reasoning	L-Q1*	Use the Mistral-7B LLM to answer and provide chain-of-thought reasoning for the GSM8K arithmetic reasoning questions, guaranteeing that the answer is an integer.	0	6	1	1	0	1	2
	L-Q2*	Use the Mistral-7B LLM to answer and provide chain-of-thought reasoning for the Date understanding dataset, guaranteeing that the answer is a valid date format.	0	6	1	1	0	1	2

shows an imputed temperature value (99) at the 4th hour, which significantly exceeds neighboring non-missing temperature values and causes an erratic spike in the temperature values.

Integrity Constraints. We wrote a total of around 30 queries to find different violations of this smoothness assumption. We investigate three of them T-Q1, T-Q2, and T-Q3 in Table 3, which describe two variations of the smoothness assumption.

The first such variant is that the difference between two continuous entries along the temporal dimension must be smooth and insignificant. One can use domain knowledge to determine a threshold to filter out those consecutive values with large differences, such as the outlier highlighted in Figure 9. In the absence of such domain knowledge, however, we formulate T-Q1 to collect all pairs of non-missing consecutive values from the entire dataset, and compute their differences. We

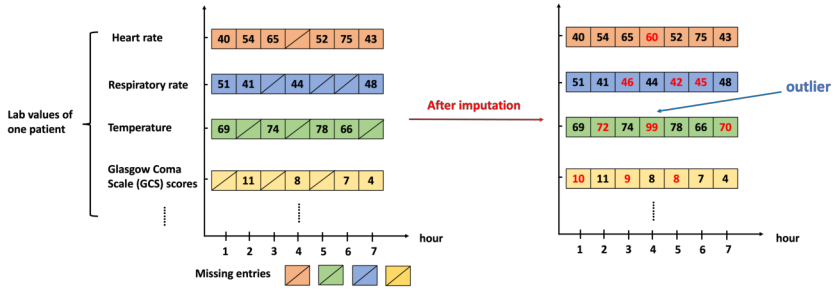


Fig. 9. A medical time series sample consisting of lab values of one patient over time. We show the original data with missing values on the left and the values imputed by SAITS on the right in red. The imputed value "99" for "Temperature" is an outlier since it deviates too far from other non-missing data for this feature.

then use the 99th percentile of all these gaps as the estimated threshold, in accordance with the method outlined in [Naik et al. 2023].

The other variant of the smoothness constraint that we specify captures the closeness between each imputed value and other entries within a time window. Specifically, we craft T-Q2 and T-Q3 to determine whether one imputed entry is an outlier or not with respect to other entries within a univariate time series. We discuss T-Q2 and T-Q3 in more detail in Section 6.2.

5.3 Image Classification

Overview and Setup. In this task, the goal is to classify the animal present in individual frames of videos captured by trap cameras. We evaluate the iWildCam dataset [Beery et al. 2020] and the predictions of a ResNet model [He et al. 2016] trained on the iWildCam training dataset.

Correctness Problems. The iWildCam dataset poses several challenges to machine learning models. We present two such problems that we investigate in more detail. First, models must have the ability to classify a frame as empty when no animal is present. As such, it is necessary for empty frames to be correctly labeled. However, there are several instances in iWildCam where empty frames are mislabeled as containing an animal. Second, the model tends to be inconsistent in its predictions. Over a sequence of frames, there is typically a single animal that moves around. However, the model sometimes predicts different animals in different frames within the same sequence.

Integrity Constraints. We wrote around 25 queries to capture various integrity constraints over the model predictions including whether nocturnal animals were predicted during the daytime, whether multiple animals were predicted across the same sequence of frames, and finding correlations between mispredicted classes and the ground truths. We analyze two of these queries.

The first, query I-Q1, aims to find sequences where the predictions by the model vary across frames, but the ground truth remains the same. The second, I-Q2, aims to find empty frames that are mislabeled to contain animals. Since the ground truth cannot be relied on, we compute the pixel differences between pairs of consecutive frames to determine frames that have no movement and thus are likely empty. We investigate I-Q2 in more detail in Section 6.3.

5.4 Text Generation

Overview and Setup. In this task, the goal is to generate text (as a sequence of tokens) given a prompt. In particular, we consider the Alpaca instruction fine-tuning dataset [Taori et al. 2023] (Alpaca for short), which consists of 52K prompts and their responses from LLMs. In this experiment, we focus on the model responses from the GPT-3.5 (`gpt-3.5-turbo`) and T5[2] models.

[2]https://huggingface.co/lmsys/fastchat-t5-3b-v1.0

Correctness Problems. Large language models (LLMs) are known to exhibit biases that can be potentially harmful [Havaldar et al. 2023; Liang et al. 2021; Zhao et al. 2017]. This can occur when the data they are trained over contain biases that the LLM may pick up on. However, typical metrics for language generation, such as the Bleu score, do not have the ability to check for them.

Integrity Constraints. We investigate how integrity constraints can be programmed using TorchQL to detect and study biases in LLMs. We wrote a total of around 50 queries to investigate various biases, and analyze two such queries in detail.

First, we write a query B-Q1 to discover the biases in the prompts of the Alpaca dataset by identifying the adjectives that are highly correlated with the occupations of farmer and engineer. This covers the class of adjective-profession biases [Kurita et al. 2019; Li et al. 2020; Nangia et al. 2020; Smith et al. 2022], where certain adjectives are biased towards certain professions (e.g. brilliant scientists). Further details and examples of this bias are provided in Section 6.4. Second, we write a query B-Q2 to quantify bias in the model response. Here, we take adjectives associated with farmers and adjectives associated with engineers to construct prompts to determine how frequently an LLM uses the farmer adjective to describe farmers and the engineer adjective to describe engineers. More details are included in Section 6.4.

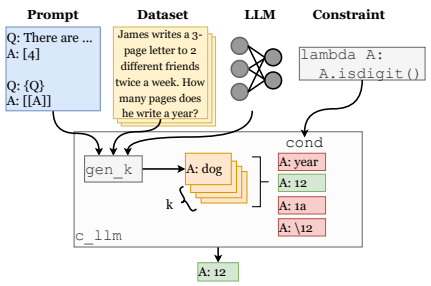
5.5 Natural Language Reasoning

Overview and Setup. In this task, the goal is to answer reasoning questions in natural language text. We consider two reasoning tasks: *date understanding*, which involves reasoning over dates and relative durations to determine the described date, and *arithmetic reasoning*, which involves solving word problems using basic arithmetic to compute desired quantities. For date understanding, we use the BigBench benchmark's date understanding task (Date) [et al. 2023] and for arithmetic reasoning we use GSM8K [Cobbe et al. 2021]. We use the top performing 7B parameter pretrained model from the Open LLM leaderboard which at the time of writing was Mistral-7B (`mistralai/Mistral-7B-v0.1`) [Jiang et al. 2023]. The prompt used for both tasks is the same prompt used by Lyu et al. [2023]. To decode the output of the LLM, we use a temperature of 0.4 along with the default parameters to the Huggingface generation API.

Correctness Problems. One of the major issues with LLMs is that the output generated is not guaranteed to conform to the conditions of the ground truths. For example, LLMs should generate valid dates in response to prompts in the date understanding task, and numbers for the arithmetic reasoning task. There have been elaborate prompting mechanisms devised to improve the reliability of LLMs such as chain-of-thought [Wei et al. 2022] and ReACT [Yao et al. 2023] and systems for programming these prompting mechanisms [Chase 2022].

Integrity Constraints. TorchQL is useful for orchestrating language model prompting due to its highly expressive user-defined functions. Placing language model inference calls inside user-defined functions allows for queries which prompt language models and process the output. We demonstrate the usefulness of this interface with a query to constrain the output of a language model similar to LMQL [Beurer-Kellner et al. 2023]. We then evaluate our prompting technique and show that TorchQL can be used to constrain LLM output while maintaining accuracy as well as LMQL on two benchmark reasoning tasks.

In order to achieve this, we build a helper query, shown in Figure 10b, whose operation is shown in Figure 10a. This query samples multiple responses from an LLM and filters out the responses which violate the given constraints. The query uses four TorchQL operations and many of the UDFs are single-lined lambda functions defined within the query specification, except for the implementation of the prompting semantics which include some use of regular expressions.



(a) Constrained generation overview.

```
def gen_k(n, dataset, prompt, k=10):
    return Query(n, base=dataset)
        .project(parse_and_prepare_prompt)
        .project(lambda prompts, togen: (llm(prompts, k),
        ↪  togen), bs=10)
        .project(parse_outputs)
        .flatten()
def c_llm(n, dataset, prompt, cond, ans, k=10):
    return gen_k(n, dataset, prompt, ans, k)
        .filter(lambda prompt, gen: cond(gen))
        .group_by(lambda prompt, gen: prompt)
        .project(lambda prompt, gens: ans(gens))
```

(b) Constrained generation with TorchQL.

Fig. 10. Using TorchQL to orchestrate prompting of language models with output constraints.

(a) (b) (c)

Fig. 11. A violation of the object consistency constraint. Figures 11a to 11c are three continuous frames from the Cityscapes dataset. A large truck is detected in the first two frames but not in the third.

We use this helper query to write around 20 queries to enforce various constraints over the Mistral-7B LLM. We analyze two of them in this paper. First, we write the query L-Q1, which performs Chain-of-Thought (CoT) [Wei et al. 2022] prompting with constraints for the arithmetic reasoning task. The second query L-Q2 similarly attempts to constrain the LLM for the date understanding task. Both queries are explored in more detail in Section 6.5.

6 CASE STUDIES

In this section, we evaluate the expressivity of TorchQL through a series of case studies for each of the above tasks. In each case study, we discuss the integrity constraints and queries from Table 3 and provide a detailed analysis of the issues discovered with those queries.

6.1 Case Study: Prediction Error Patterns in Self-Driving Object Detectors

Object Consistency Constraint. Section 2 describes a version of the object consistency constraint (S-Q1) specialized over pedestrians. S-Q1 finds 6,264 objects that are detected in two consecutive frames and disappear in the third. To evaluate the effectiveness of this query, we look at how well it finds model mispredictions by focusing on the 500 frames with ground truth bounding box labels provided by the Cityscapes dataset. Of the tuples of three frames (from S-Q1), where an object disappears in the third frame, there are 113 frames with labeled ground truth bounding boxes. In 71 out of these 113 frames, the query successfully detects an object that was missed by the model. Thus, this query has 62.83% precision. In total, there are 1388 mispredicted objects, so the query has a recall of 5.12%.

Overall, S-Q1 has good precision but low recall since the violations of just this integrity constraint alone cannot capture a large percentage of all the model failures. On the other hand, this constraint does capture real safety-critical issues. For example, as Figure 11 shows, as the camera vehicle (i.e., the vehicle from which the video is recorded) gradually approaches the white van parked on the

(a) (b)

Fig. 12. A violation of the high-velocity constraint where Figures 12a and 12b are two continuous frames from the Cityscapes dataset. The predicted bounding boxes in these two frames are inconsistent in their positions and sizes, thus exaggerating the speed of the object.

left, the object detector suddenly fails to detect it (see Frame 3) even though it is predicted in the previous two frames. Considering the short distance between the two vehicles, such behavior by an object detector deployed in an autonomous vehicle could potentially result in an accident.

High-velocity constraint. Building on top of S-Q1, we construct S-Q2 for computing the speed of objects detected in three consecutive frames and filtering them. The speed of an object is calculated by estimating the distance the object moves across consecutive frames. If an object has an unusually high speed, it may indicate an inaccurate object detector prediction. For example, the bicycle shown in Figure 12 is the object with the highest speed computed by S-Q2. This is primarily caused by abrupt variations in the bounding box size in the two consecutive frames. As Figure 12 shows, the bounding box in frame 1 only covers a fraction of the bicycle. On the other hand, the bounding box in frame 2 correctly identifies the entirety of the bicycle. This results in the bicycle appearing to move an abnormally large distance within the span of two frames. The following shows the full TorchQL query to represent this integrity constraint.

```
Query('fast_vehicles', base = 'three_adj_matches')
    .filter(get_vehicles)
    .project(get_velocities)
    .filter(lambda seq_id, frames, boxes1, boxes2, velo: velo > VELO_CUTOFF)
    .group_by(seq_id_frame_id)
```

This query operates over the 'three_adj_matches' table which consists of objects found across three consecutive frames. This table is constructed from the same query as 'temporal_consistency' from Figure 5a, just without the last filter operation. The VELO_CUTOFF constant in the query is selected based on the 99th percentile of speeds to be 51.62. This query selects 1,172 pairs of objects over 395 frames. Of these frames, 36 have labeled ground truth bounding boxes and each object selected in the ground truth appears in the set of mispredictions. Thus, this query has a precision of 100% and a recall of 2.59% over mispredictions.

6.2 Case Study: Imputed Value Monitoring for Time-Series Healthcare Data

We craft queries T-Q2 and T-Q3 to specify constraints on the smoothness assumptions in a univariate time series. We only focus on T-Q2 in this section as T-Q3 just differs in the way the threshold for detecting outliers is determined. T-Q2 joins the table containing imputation results with the ground truth table to obtain all the non-missing values and imputed values. We then employ a user-defined function to filter out imputation outliers over each univariate time series of every sample. This is accomplished by leveraging a threshold determined through established statistical methods for outlier detection [Tukey et al. 1977].

We illustrate this outlier detection method on a patient's respiratory rate from the Physionet-2012 Challenge dataset shown in Figure 13. In this univariate time series, the non-missing respiratory rate values are denoted by blue dots while the imputed values (including outliers) are denoted by

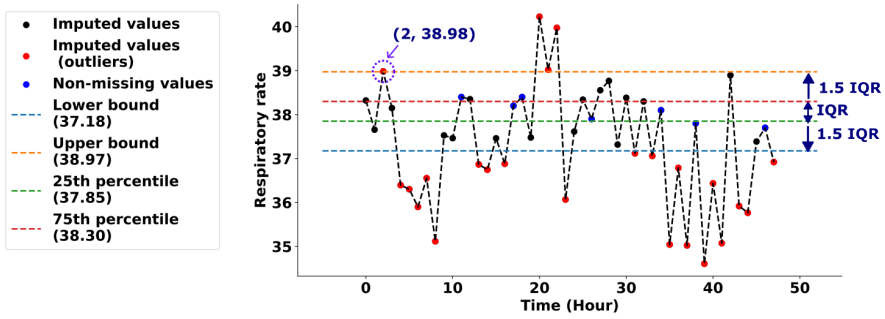


Fig. 13. Finding imputation errors as outliers to the range of expected respiratory rates for a patient. Using the non-missing values of respiratory rate, upper and lower bounds are calculated and all imputed values outside the lower-to-upper bound range are selected as imputation errors (shown as red dots).

black or red dots. The outliers (denoted by red dots) are those imputed values outside the lower or upper bound shown in Figure 13. To determine these two bounds, we first calculate the interquartile range (IQR) of all the non-missing values, which is the range between the 25th percentile (LQ) and 75th percentile (UQ) of those values. They are then derived using the following formula:

$$\text{Lower bound} = LQ - 1.5 \cdot \text{IQR}, \quad \text{Upper bound} = UQ + 1.5 \cdot \text{IQR}$$

As Figure 13 suggests, 24 outliers are identified from the imputed values with the constraint specified by T-Q2, most of which are all visually far away from their nearest non-missing values.

We further report the *recall* of T-Q2 over 20% randomly held-out non-missing entries, i.e., the portion of the entries with imputation errors covered by T-Q2. Ideally, T-Q2 covers a significant proportion of the total imputation errors allowing us to find where the model makes its worst errors and potentially allow for highly effective solutions. Hence, over the 20% held-out non-missing entries, we evaluate the ratio of the imputation errors covered by T-Q2 to the total imputation errors, denoted as *imputation error*. Six other queries were written to detect outliers over different variables using similar smoothness assumptions, and so we do the same to evaluate the recall and imputation error fall all seven queries.

Ideally, both *recall* and *imputation error* approach 100%. The recall and imputation error are 40.15% and 54.27% respectively for all seven smoothness assumption queries, while they are 26.14% and 34.23% respectively just for T-Q2. The intermediate-level recall and imputation error of these queries justifies their validity and usefulness.

It is also worth noting that queries written in TorchQL can reveal model prediction issues that cannot be captured by standard model performance metrics such as the Mean Square Error (MSE). For example, for the entry at hour 2 in Figure 13 whose ground truth is 38.95, the imputed value at this entry is 38.98, which is marked as an outlier by T-Q2 since it is above the upper bound (38.97). However, the MSE metric cannot differentiate the cases where the imputed value is 38.98 or 38.92 for this entry since the difference between either value and the ground truth is the same. On the other hand, the imputed value 38.92 won't be flagged as a violation of T-Q2. One can therefore use T-Q2 to prefer an imputed value of 38.92 over 38.98, while the MSE is not fine-grained enough to determine which value is better.

6.3 Case Study: Mislabeled Data in iWildCam

We notice that in the iWildCam dataset, there are several cases where animals are present in the first few frames of a trap camera video, but then exit the frame. Once the animal exits, the

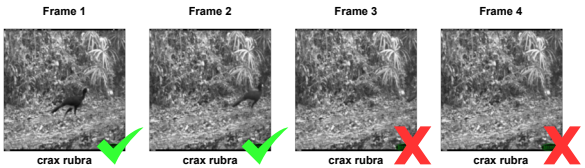
```
db = Database()
db.register(train_data, 'wilds_train')

imvar = Query('imvar', base='wilds_train')
    .group_by(lambda im, lb, m: m[1].item())
    .project(lambda seqid, r: [(seqid,
      i-1, r[i-1][0], r[i-1][1], r[i-1][2],
      i, r[i][0], r[i-1][1], r[i][2],
      diff(r[i-1][0], r[i][0], 0.1, 0.15)
    ) for i in range(1, len(r))])
    .flatten()

imvar(db)
```

(a) Finding pixel differences between consecutive frames.

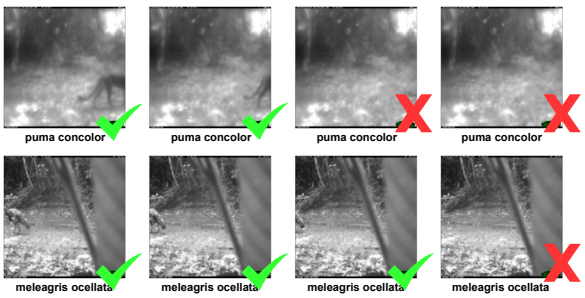
```
def is_empty(*row):
    diff = row[-1]
    mean = get_non_zero_mean(diff).item()
    return mean <= 0.13 and mean > 0

nonempty = Query('nonempty', base='imvar')
    .filter(lambda *args: not (
      args[3] == 0 and args[7] == 0))
non_empty(db)

mislabeled = Query('mislabeled',
      base='nonempty').filter(is_empty)
mislabeled(db)
```

(b) Finding mislabeled frames.

Fig. 14. Query to find mislabeled frames in the iWildCam dataset.



(a) An example of mislabeled data found while manually exploring the iWildCam training data.



(b) Mislabeled data identified by the `mislabeled` query from Figure 14b. There are 280 such instances.

Fig. 15. Examples of mislabeled data from the iWildCam training dataset. Frames marked with a check are correctly labeled. Frames marked with a cross are those without an animal but still labeled non-empty. The correct label in such cases should be `empty`.

subsequent frames should be labeled as empty. However, in many cases, these frames are labeled in the ground truth as if the animal is still present. We show an example of this in Figure 15a, where the label for each frame is `leopardis pardalis`, despite the animal not being present in frame (d).

In most cases when animals are present in the frame, they tend to move around, creating a significant difference between the pixel values of consecutive frames. We therefore program the integrity constraint that any pair of consecutive frames with a small difference in their pixel values are most likely empty, and should therefore be labeled as such.

We construct this integrity constraint over a sequence of three queries shown in Figure 14. We begin by finding the difference in pixels over all pairs of two consecutive frames using the `imvar` query (Figure 14a). The `nonempty` query is then used to filter to frames not labeled as empty. We

Fig. 16. Visualizations of the adjectives associated with *farmer* and *engineer* from the Alpaca dataset are shown on the left. Biased responses of GPT-3.5 when queried with the circled adjectives are on the right.

then determine a threshold to distinguish the pixel differences between frames with empty labels and the ones with non-empty labels. We use this threshold (0.13) to build the `mislabeled` query (Figure 14b) to detect visually empty frames incorrectly labeled as nonempty. More details about the queries are provided in Appendix B.

Overall, we detected 280 instances in the training data out of 69,972 pairs of consecutive frames. We show some examples of sequences containing such mislabeled data in Figure 15b. Since these mislabels occur within the ground truth, we manually inspect 20 random instances and find three false positives. In other words, there are only three cases out of the 20 where the frame satisfied this condition but actually had an animal in it. This suggests the constraint is accurate in capturing empty frames that are mislabeled as non-empty.

6.4 Case Study: Bias Discovery in LLMs

We write integrity constraints in TorchQL to discover adjective-profession biases in the Alpaca instruction-tuning dataset [Taori et al. 2023].

Detecting Biases in Datasets. We first discover biases in the Alpaca dataset with query B-Q1:

```
Query('farmer_adj', base='alpaca')
    .project(lambda instr, inp, outp: [instr, inp, outp, extract_noun_adj_pairs(outp)])
    .filter(lambda instr, inp, outp, pairs: len(pairs) > 0)
    .project(lambda instr, inp, outp, pairs: pairs)
    .flatten()
    .project(lambda noun, adj: [noun.lower(), adj.lower()])
    .group_by(lambda noun, adj: noun)
    .filter(lambda noun, adj: noun == "farmer")
```

Intuitively, this query finds all the adjectives used to describe the noun *farmer* in the Alpaca dataset. We find nouns and adjectives using a natural language parsing model from the Spacy library [Honnibal and Montani 2017]. This library is used by the `extract_noun_adj_pairs` function called within the query. We then write a similar query to find adjective-profession biases for "engineer". We visualize adjectives associated with *farmer* and *engineer* in the Alpaca dataset in Figure 16.

The figure shows a clear difference between adjectives associated with farmers and engineers. We see undesirable differences tied to factors like age, race, and economic status (*old*, *African*, and *poor* for farmers), and variations in perception, such as engineers associated with *responsible*.

Monitoring Biases in Model Outputs. Now that we know the biases that exist in the Alpaca dataset, we program integrity constraints for monitoring the biases in the outputs of LLMs. We do so by building on top of the 'farmer_adj' and 'engineer_adj' queries. We specify that a language model should not favor exclusively using adjectives associated with farmers when describing farmers,

or adjectives associated with engineers when describing engineers. To do this, we prompt LLMs like GPT-3.5 with the question "Answer with one word. The person is a farmer. Are they more likely *farmer_adj* or *engineer_adj*?", in which *farmer_adj* and *engineer_adj* are a pair of farmer and engineer adjectives. We automate this process with the following TorchQL queries which prompt GPT-3.5 in the above manner and then compute how often the biased answer is chosen:

```
Query('gpt35_bias_answers', base='farmer_adj')
    .join('engineer_adj', key=lambda *args: 1, fkey=lambda *args: 1)
    .project(lambda farmer_adj, engineer_adj: [
        f"Answer with one word. The person is a farmer. \
        Are they more likely {farmer_adj} or {engineer_adj}?" ... ])
    .flatten()
    .project(lambda prompt: gpt35(prompt))
    .project(lambda prompt, answer: [*parse_prompt(prompt), parse_response(answer)])
    .group_by(lambda target_job, adj1, adj2, ans: (target_job, frozenset({adj1, adj2})))
    .project(lambda key, rows: [key[0], key[1], [row[3] for row in rows]])
Query('gpt35_bias_chosen', base='gpt35_bias_answers')
    .project(lambda job, adjs, res: [[job, adjs, w] for w in res if w in adjs])
    .flatten()
```

We collectively refer to the above queries as B-Q2. Note that the above queries were querying the GPT-3.5 LLM within the query itself using the `project` function. We run a similar query for the T5 language model as well, and then compare the biases exhibited by both models. Out of all the model responses that contain an adjective correlated with farmers or engineers, T5 selects the biased adjective 58.0% of the time and GPT-3.5 selects the biased adjective 59.6% of the time. For GPT-3.5, these biased responses cover 23 of the 27 farmer-associated adjectives and 34 of the 40 engineer-associated adjectives. We show an example of this on the right of Figure 16, where GPT-3.5 consistently generates biased adjectives. Similarly, the biased responses of T5 cover 11 of the 14 farmer adjectives and 15 of the 20 engineer adjectives. Note that a perfectly unbiased model would select the biased adjective 50% of the time. This indicates that this query is able to identify biased responses from LLMs like GPT-3.5 and T5.

6.5 Case Study: Constraining the Output of Language Models

The query described and shown in Figure 10 implements a general constrained prompting interface. Within the UDF `parse_and_prepare_prompt`, we implement a simple prompting interface based on that of LMQL where text in a prompt surrounded by curly braces, such as "{question}", is replaced by elements from a table, and text appearing in double square brackets, such as "[[answer]]" is generated by the language model. A benefit of using TorchQL in this case is that it provides in-house support for batching. Since the language model takes batched inputs, as do most machine learning models, we use a batch size of 10 shown in Figure 10b by passing the `bs=10` argument to TorchQL's `project` operation for automatic batch-wise parallelism. Performing similar batching with LMQL requires using Python's asynchronous functionality and setting up a semaphore to limit the number of concurrent processes.

Unlike LMQL, we cannot perform any constraint-specific optimizations, since TorchQL allows constraints to be arbitrary Python functions. Despite this, we show that for the tasks of arithmetic problem solving and date understanding, the TorchQL queries still improve constraint satisfaction without hurting accuracy. Another benefit of our approach is that we can use black-box functions, including other models or even the Python interpreter. Using the `c_llm` function, we define 2 queries which perform constrained CoT prompting on the two tasks. The query performing constrained CoT prompting on arithmetic reasoning is L-Q1 and the query performing constrained CoT on date understanding is L-Q2 from Table 2. Query L-Q1 is shown on the right of Figure 17. L-Q2 is the same query with a modified prompt for the date understanding task and the constraint:

```
lambda args: len(args['reasoning']) > 10 and re.match(r"\d\d/\d\d/\d\d\d\d", args['answer']).
```

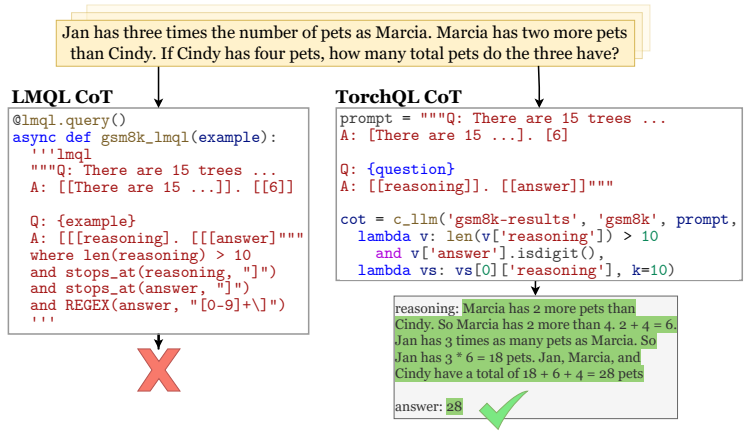


Fig. 17. Use of TorchQL for constrained generation on the GSM8K arithmetic reasoning task compared to LMQL. The two methods can result in different responses on the same prompt. LMQL fails to produce any constraint-satisfying output while TorchQL results in output which is a valid integer and the correct answer.

Table 4. Performance results for using TorchQL for language model prompting. Validity is the percent of all samples with a constraint-satisfying answer and accuracy is the task accuracy.

		Date Understanding			Arithmetic Reasoning		
Method	Constraints	Valid	Accuracy	Time (m)	Valid	Accuracy	Time (m)
CoT	No	96.75	52.85	**2.6**	81.88	37.83	**9.35**
LMQL CoT	Yes	98.37	52.85	68.03	89.92	38.67	226.12
TorchQL CoT	Yes	**99.73**	**53.39**	5.72	**97.80**	**44.28**	20.68

The full query is provided in Appendix B. We compare the TorchQL query L-Q1 with its LMQL CoT counterpart in Figure 17. For the question shown in Figure 17, LMQL fails to produce any valid response while our method with TorchQL produces a valid and correct response. This failure of LMQL is potentially caused by the decoding algorithm which may sometimes generate long streams of text without returning a valid answer.

The results for using TorchQL to prompt an LLM for the GSM8K and Date dataset are in Table 4. The 'Valid' column shows the percent of samples where a constraint-satisfying answer is produced, and the 'Accuracy' column shows the task accuracy of the results. We see that this LLM constraint method with TorchQL successfully constrains the LLM, resulting in better generation validity than LMQL for both tasks, as well as higher task accuracy. Finally, using TorchQL for constrained generation is up to 11x faster than using LMQL even though we used the same batch size and model, but only about 2x slower than unconstrained CoT.

7 QUANTITATIVE EVALUATION

The previous section studied the expressiveness of TorchQL across different use-cases. We now conduct quantitative experiments to evaluate the efficiency and conciseness of TorchQL queries compared to their counterparts in baseline systems.

7.1 Setup

We consider three baselines for this set of experiments: native Python, the standard language used for machine learning applications; NumPy [The NumPy Team 2024], which is an efficient array

Table 5. Running time (seconds) for each system (smaller is better). For each query, the quickest is marked in bold, while the next quickest is underlined. All queries had a timeout set to 10 minutes.

Query	Running time (seconds)				
	Python	Numpy	ArangoDB	Pandas	TorchQL
S-Q1	**43.32±0.52**	—	—	TO	44.51±0.42
S-Q2	**11.27±0.87**	—	TO	32.89±0.20	16.83±0.97
T-Q1	0.11±0.02	0.12±0.02	TO	0.13±0.03	**0.11±0.04**
T-Q2	2.59±0.05	**1.35±0.02**	9.32±1.66	5.76±0.35	2.65±0.38
T-Q3	0.90±0.05	**0.56±0.00**	57.38±1.89	1.93±0.55	0.95±0.08
I-Q1	**0.11±0.05**	—	0.24±0.13	0.95±0.08	0.18±0.01
B-Q1	**0.012±0.02**	—	—	0.085±0.03	0.016±0.01

Table 6. Conciseness comparison (smaller is better). For each query, we show the count of tokens of the query in each system. The smallest number is marked in bold and the next smallest is underlined.

Query	Conciseness (number of tokens)				
	Python	Numpy	ArangoDB	Pandas	TorchQL
S-Q1	183	-	-	175	**113**
S-Q2	109	-	207	113	**97**
T-Q1	110	111	171	186	**97**
T-Q2	99	99	174	120	**95**
T-Q3	148	175	167	183	**139**
I-Q1	75	-	64	**43**	55
B-Q1	104	-	-	98	**89**

manipulation Python library; ArangoDB [ArangoDB 2023], which is a document-oriented database and effectively an in-memory version of MongoDB; and Pandas [pandas development team 2020], an in-memory data-analysis Python library. Since the OMG [Kang et al. 2018] model assertion system uses Python as its query language, our Python comparison also serves as a comparison against OMG.

For each query in Table 2, with the exceptions of I-Q2, B-Q2, L-Q1, and L-Q2, we implement the query in TorchQL as well as in each baseline system. Each query evaluated is deterministic and returns the same results each time it is run across all baselines. We include all queries in Appendix C. Note that it is infeasible to write certain queries in certain baseline systems. For instance, queries S-Q1 and B-Q1 cannot be written in ArangoDB because the user-defined functions that ArangoDB can support are quite limited. Also, only the T-Q1, T-Q2, and T-Q3 queries can be written using NumPy operators. This is because the other queries largely involve string and list operations, which are not supported by purely NumPy operators.

In order to ensure a fair evaluation, we made the best effort to optimize the code within the style typical of each framework. For instance, we optimize the Python code so that its algorithmic complexity matches that of TorchQL. We also attempt to use as few DataFrame operations in the Pandas implementations and as few database operations in the ArangoDB versions as possible. We compare the different implementations of each query to evaluate its efficiency (by measuring the running time) in Table 5, and the conciseness (by measuring the number of tokens) in Table 6.

7.2 Results

Efficiency. In all cases, TorchQL queries are comparable or faster than their Python counterparts. This is expected, since TorchQL itself is written in Python. Furthermore, each Python query is

Table 7. Running time (in seconds) of each query operation for queries written in TorchQL and Python. For each query, we report the total time taken for each type of operation to be executed while running the query.

Query		Query Operators					
		joins	*projects*	*filters*	*groupings*	*orderings*	*flattens*
S-Q1	TorchQL	0.28±0.01	43.15±0.26	0.14±0.01	—	—	0.01±0.00
	Python	0.10±0.02	41.91±0.08	0.13±0.01	—	—	0.01±0.00
S-Q2	TorchQL	4.63±0.73	11.10±0.65	2.96±0.03	—	—	—
	Python	0.64±0.02	7.87±0.42	3.54±0.03	—	—	—
T-Q1	TorchQL	0.01±0.00	0.11±0.01	—	—	—	—
	Python	0.00±0.00	0.11±0.03	—	—	—	—
T-Q2	TorchQL	0.01±0.00	2.36±0.09	—	—	—	0.02±0.02
	Python	0.01±0.00	2.32±0.03	—	—	—	0.02±0.02
T-Q3	TorchQL	—	0.95±0.08	—	—	—	—
	Python	—	0.90±0.05	—	—	—	—
I-Q1	TorchQL	—	0.02±0.00	0.01±0.00	0.08±0.01	0.01±0.00	—
	Python	—	0.02±0.00	0.00±0.00	0.08±0.01	0.01±0.00	—
B-Q1	TorchQL	—	0.01±0.00	—	0.00±0.00	—	—
	Python	—	0.00±0.00	—	0.00±0.00	—	—

optimized to match the algorithmic complexity of the TorchQL version. The running time overhead introduced by TorchQL's querying abstractions is negligible for the most part, being less than one second for all queries except S-Q2, where it is around five seconds. This is still a small difference given the scale of the data for S-Q2. We further break down these running times by each query operator in Table 7. There is a trade-off for optimizing the Python code to such an extent: doing so drastically increases the required number of tokens, as evident from Table 6. This also results in the process of iterative debugging becoming more cumbersome. On the other hand, TorchQL can scale better than non-optimized versions of the Python queries. For instance, the non-optimized S-Q1 query written in Python (Figure 5b), which is more representative of how users may write Python queries to iteratively debug their models, times out after 10 minutes.

In the case of NumPy, the query for T-Q1 has almost the same running time as the Python and TorchQL versions. On the other hand, the NumPy versions of T-Q2 and T-Q3 run faster than their corresponding Python or TorchQL counterparts. This is due to the "quantile" operation needed for defining those queries. For the Python and TorchQL versions, we use Pytorch's "quantile", while we use its NumPy counterpart for queries written using NumPy. The NumPy implementation of "quantile" is faster than the PyTorch implementation, resulting in a quicker running time for the NumPy queries. However, like the Python queries, the NumPy ones require more tokens than the TorchQL versions as Table 6 suggests.

Pandas times out on S-Q1 after 10 minutes due to the scale of the data, and is several seconds slower than TorchQL in most other cases. The primary factor contributing to the inefficiency is the rigorous schema requirement of Pandas. Pandas is optimized for cases where the DataFrames have a fixed structure with primitive datatypes in cells. As a result, additional operations are needed to explicitly produce new attributes essential for the downstream operations but not in the schema. Since these operations tend to modify the DataFrame itself, they introduce substantial overhead.

ArangoDB is even slower and times out on S-Q2 and T-Q1. The reason is that for queries like S-Q2 which join two datasets on attributes that are not indexed, ArangoDB has to perform full nested scans on the datasets for the join operations (see Figure 19d where the join is conditioned over

`item1.frames[0] == item2.frames[0]` but `frames[0]` is not indexed). On the other hand, TorchQL can construct indexes over such newly created attributes on the fly without extra operations.

Overall, TorchQL achieves at least a 13x and three order-of-magnitude speed-ups in with respect to Pandas and ArangoDB respectively in the best case (S-Q1 for Pandas and T-Q1 for ArangoDB). Note that Pandas timed out for S-Q1, which is why the 13x speed up in this case is a lower estimate.

Conciseness. Table 6 shows the conciseness metric evaluated on each query written in different systems. TorchQL significantly reduces the number of tokens needed for writing a query (by up to 40%) with respect to native Python. This can substantially reduce users' effort in the process of iteratively writing and refining queries for specifying appropriate integrity constraints. Furthermore, TorchQL is more concise than Pandas for all cases except I-Q1, and more concise than ArangoDB in general. This is because these baselines require substantial data wrangling so that their rigid schemas can support arbitrary Python objects, which is not needed for TorchQL.

8 USER STUDY

We conduct a small-scale user study with 10 participants to validate the usability of TorchQL for programming integrity constraints over complex forms of data.

8.1 Setup

Participants. Of the total of 10 participants in the user study, one was a systems engineer, three were undergraduate students, and six were graduate students. Eight participants had experience with machine learning through research or classes and all ten were proficient in Python programming. Only three had taken a course on databases or query languages. None of the participants were the authors of this paper or involved in developing TorchQL in any way.

Tasks. For the user study, each participant was provided with a tutorial of TorchQL in a Jupyter notebook (an interactive Python environment) and then asked to write three TorchQL queries (tasks T1, T2, and T3) of increasing complexity over the iWildCam dataset. All three tasks, detailed in Appendix D.2, focused on finding model prediction errors. Task T1 required finding mispredicted frames while referring to the labels, while T2 required finding the time of day when frames are most and least likely to be mispredicted. For both these tasks, participants were allowed to make use of the supplied ground-truth labels. Task T3, on the other hand, was more open-ended in nature, requiring them to find sequences of frames with at least one misprediction without using the supplied ground-truth labels. No time limit was enforced on any task, though we measured the approximate time it took users to complete each task. We also measure the precision and recall of the responses of the users for the third task.

8.2 Results

Table 8 shows the amount of time (in minutes) taken by each user to complete each task, as well as the precision, recall, and F1 score of each user's solution for task T3. Overall, users completed the first two tasks in an average of 3.42 and 20.45 minutes respectively. A total of seven of the 10 users completed the third and most complex task, where they had to find model mispredictions without using ground-truth labels. Their queries had an average precision of 88.0% and recall of 71.0% for finding model mispredictions, compared to our solution with a precision and recall of 91.0% and 78.0% respectively, and were written in an average of 54.33 minutes. The large difference in average time taken for the three tasks reflects their differing complexity. These aggregate numbers show the usability of TorchQL despite the users' lack of prior familiarity.

Looking at individual users' queries reveals where people had difficulty and general trends in the use of TorchQL. The first task required a single `filter` operation and all users correctly utilized

Table 8. User Study Results. For each participant, we show the amount of time (in minutes) taken by the user to complete each task, as well as the precision and recall for their response to task T3.

Participant	Tasks			Performance Metrics	
	T1	T2	T3	Precision	Recall
P1	2	60	90	1.0	0.7
P2	5	35	55	—	—
P3	2	16.5	19	0.91	0.78
P4	5	15	45	0.91	0.78
P5	1	15	120	0.6	0.4
P6	9	12	15	—	—
P7	2	10	45	0.91	0.78
P8	5	20	60	—	—
P9	3	20	40	0.91	0.78
P10	0.16	1	5	0.91	0.78
Mean	3.42	20.45	49.4	0.88	0.71
Minimum	0.16	1	5	0.6	0.4
Maximum	9	60	120	1	0.78

this operation to perform the first task. The second task required the use of a `group_by` followed by an `order_by`. Seven users successfully used these two operations and composed them in the correct way. The other three correctly used the `group_by` operation, but resorted to manually iterating through the table with a `for` loop to determine the largest and smallest element of the table.

Of the seven responses for the third task, five were equivalent to our solution. These solutions, though syntactically different, were all programmed to catch violations of the same integrity constraint: each sequence of images contains only one species of animal. We show a comparison between one such solution from participant P4 (left) and our expected solution (right):

```
Query('T3_P4', base='wilds_pred')
    .project(lambda img, label, md, pred:
        [md[1].item(), img, md, pred])
    .group_by(lambda seqid, img, md, pred: seqid)
    .filter(lambda seqid, data:
        len(set(e[3] for e in data)) > 1)
```

```
Query('T3_ours', base='wilds_pred')
    .project(lambda img, label, md, pred:
        [md[1].item(), pred])
    .group_by(lambda seq, pred: seq)
    .filter(lambda seq, preds:
        len(set(preds)) > 1)
```

Both queries first project out the sequence ID of each image (corresponding to `md[1]`), then group the objects of the resulting tables by the sequence IDs, and finally filter the instances with more than one unique prediction. The differences occur in the definition of the `filter` lambda function due to the structure of the intermediate table input to that operation. In our case, each row in the grouped table contains just the sequence ID and the prediction of each image in that sequence, while in P4's case, it contains the image and metadata as well.

Among the two remaining queries, participant P1's query achieved better precision than our solution but had worse recall, while P5's query performed worse over both metrics. We show snippets of their queries here and their full versions in Appendix D.1:

```
def conflicting_pred(lst):
    ...
def get_sequence_preds(lst):
    return [item[3] for item in lst]

Query('T3_P1', base='wilds_pred')
    .group_by(lambda _1, _2, md, _3: md[1].item())
    .filter(lambda seq, lst: conflicting_pred(
            get_sequence_preds(lst)))
```

```
Query('T3_P5', base='wilds_pred')
    .group_by(lambda _, __, md, ___: md[1].item())
    .cols(lambda seqid, rows: (
        seqid,
        len([row for row in rows if ... ]) > 1,
        len(rows), rows))
    .filter(lambda seqid, has_incorrect, _, __:
        has_incorrect)
```

Participant P1's query, shown above on the left, flagged instances where images in each sequence have inconsistent predictions where no prediction is "empty". As seen in Table 8, this query avoids any false positives, i.e. only flags sequences with at least one erroneous prediction, but has more false negatives than our expected solution. On the other hand, P5's query flags sequences based on the time, day, and month when those sequences were captured. This hypothesis is not ideal, since it results in significantly more false positives and negatives compared to the expected solution.

9 RELATED WORK

Databases for Machine Learning. Techniques from databases have been used to enhance various aspects of machine learning. Ideas from operation scheduling have been used to select appropriate hardware [Mirhoseini et al. 2017] or find semantically equivalent but more efficient operations to optimize deep learning computations [Jia et al. 2019]. Recomputing and swapping techniques from databases have been adopted to manage memory in deep learning frameworks [Pleiss et al. 2017; Wang et al. 2018]. Compression techniques and communication frameworks have been proposed to accelerate parameter servers in distributed training of large models [Goyal et al. 2017; Jiang et al. 2017]. These approaches focus on the speed and efficiency of deploying models.

Integrity Constraints in Machine Learning. Integrity constraints have been used in machine learning for data analysis and cleaning, model debugging, verification, and data generation. For instance, "unit testing" data [Schelter et al. 2019] is proposed to assess data quality in data analysis and native Python assertions have been employed to capture model output bugs [Kang et al. 2018]. In addition, there has been extensive research in neural model verification [Liu et al. 2021], which focuses on verifying whether a given input-output constraint holds for a neural network. Finally, data generation languages such as Scenic [Fremont et al. 2019] construct data to test the integrity of a machine learning system or to describe correct or incorrect model behaviors [Kim et al. 2020].

Finding Errors in Machine Learning Models. Various tools have been developed to identify errors in the machine learning systems, which can be operated in either passive or active mode. In passive mode, tools can either generate test samples to trigger model errors [Tian et al. 2018] or produce explanations to assist users in understanding the model's prediction process. These explanations can be in various forms, such as the coefficients of the sparsified neural network layers [Wong et al. 2021], extracted robust features [Singla and Feizi 2021; Singla et al. 2021], and influence functions [Ilyas et al. 2022; Koh and Liang 2017; Salman et al. 2022], which can then be compared against domain knowledge to discover model prediction errors. They can even be integrated into the subsequent model debugging loop [Anders et al. 2022; Bhadra and Hein 2015; Cadamuro et al. 2016; Kulesza et al. 2015]. Apart from that, users can actively provide rules [Kang et al. 2018] based on their knowledge to verify the correctness of model predictions. Except for the tools for finding errors in general machine learning systems, some tools have emerged for specific settings such as federated learning systems [Augenstein et al. 2019].

Query Languages in Other Domains. Query languages aim to interact with data in a declarative manner, In the past few decades, people have dedicated to developing query languages for retrieving and manipulating data from databases, such as SQL (Structured Query Language)[Codd 1970] for relational databases, and MQL [Banker et al. 2016], SPARQL [Herman 2013] and XQuery [Kilpeläinen 2012] for non-relational databases. As data sizes continue to grow, distributing data across multiple servers becomes imperative, driving the development of new programming interfaces for distributed systems, such as MapReduce [Dean and Ghemawat 2008] and Spark [Zaharia et al. 2010]. In addition, those query languages have also been extended to support streaming data. For instance, T-SQL[tsq 2022] extends SQL and supports streaming computations on relational databases while Spark streaming [Salloum et al. 2016] processes streaming workloads in distributed systems. However,

developing a programming language for capturing integrity constraints in machine learning pipelines has not been extensively studied yet in literature.

Synthesizing Programs in Query Languages. It is worth noting that some prior works have studied how to incorporate relational operators into imperative code for performance improvement. For instance, [Cheung et al. 2013] automatically extracts relational operations and synthesizes SQL queries from application code such that the underlying database optimizers can be utilized for performance enhancement. In addition, [Mariano et al. 2022] proposed a neural-guided synthesis algorithm to automatically incorporate the functional APIs into existing imperative code for parallel computations in the distributed computing environment. A less relevant work along this line is [Smith and Albarghouthi 2016], which proposed an efficient algorithm and tool for automatically synthesizing MapReduce-style distributed programs from input–output examples. Therefore, by borrowing ideas from these prior works, one future extension of TorchQL would be automating the process of transforming existing Python queries to TorchQL ones, thus reducing users' programming efforts on characterizing integrity constraints in machine learning.

10 LIMITATIONS AND FUTURE WORK

We anticipate several venues to further optimize TorchQL. First, TorchQL does not take advantage of multi-core systems with the exception of loading data into the database. However, since TorchQL follows the general structure of map-reduce pipelines, the structure of the system makes it highly amenable to parallelism by integrating multiprocessing components into TorchQL's abstractions.

Second, TorchQL's database is in-memory. This means that any data being queried over is limited by the size of the RAM of the system, limiting the ability of TorchQL to query over larger amounts of data. Traditionally databases are not in-memory to prevent this problem with scalability, and load chunks of data as needed, but this results in a loss of performance during the I/O operations associated with reading from and writing to disk.

Third, TorchQL is currently a new library and it is therefore harder to synthesize TorchQL queries than typical Python. However, initial studies have shown some initial success for synthesizing TorchQL queries via few shot prompting using GitHub's Copilot and GPT-4. To help with the interactive nature of debugging, additional support for synthesis, such as the ability to synthesize UDFs or just the query structure is an area of future work.

Finally, while TorchQL queries support recursive UDFs, supporting a fixed-point table operator would allow for optimizations such as reusing materialized database views [Chirkova and Yang 2012]. Further research can investigate optimizations from the literature on object-relational data models, like reordering query operators [Garcia-Molina et al. 2008], caching intermediate results [Pirahesh et al. 1992], exploring column stores [Abadi et al. 2009], and building more advanced indices [Lemire et al. 2010].

11 CONCLUSION

We introduced a new framework TorchQL for programming and checking integrity constraints in machine learning applications. We showed that TorchQL is able to help find errors in several tasks across various domains up to 13x faster than other querying systems while being up to 40% more concise than regular Python programs. We also validated the usability of TorchQL via a user study. In the future, we intend to extend TorchQL to further improve performance by parallelizing operations and to support synthesizing queries from natural language descriptions.

DATA-AVAILABILITY STATEMENT

The code for TorchQL along with the data are made available as an artifact published at https://zenodo.org/records/10723160 [Naik et al. 2024].

ACKNOWLEDGEMENTS

We are thankful to the anonymous reviewers for their insightful feedback that helped to improve the paper, as well as the participants of the user study. This research was supported by NSF award #2313010, an NSF Graduate Research Fellowship, and a Google PhD Fellowship.

REFERENCES

2022. Stream Analytics Query Language Reference. https://learn.microsoft.com/en-us/stream-analytics-query/stream-analytics-query-language-reference

Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1664–1665. https://doi.org/10.14778/1687553.1687625

Abubakar Abid, Maheen Farooqi, and James Zou. 2021. Persistent anti-muslim bias in large language models. In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*. 298–306.

Christopher J Anders, Leander Weber, David Neumann, Wojciech Samek, Klaus-Robert Müller, and Sebastian Lapuschkin. 2022. Finding and removing Clever Hans: using explanation methods to debug and improve deep models. *Information Fusion* 77 (2022), 261–295.

ArangoDB. 2023. ArangoDB Query Language (AQL) Introduction: ArangoDB Documentation. www.arangodb.com.

Sean Augenstein, H Brendan McMahan, Daniel Ramage, Swaroop Ramaswamy, Peter Kairouz, Mingqing Chen, Rajiv Mathews, et al. 2019. Generative models for effective ML on private, decentralized datasets. *arXiv preprint arXiv:1911.06679* (2019).

Kyle Banker, Douglas Garrett, Peter Bakkum, and Shaun Verch. 2016. *MongoDB in action: covers MongoDB version 3.0.* Simon and Schuster.

Sara Beery, Elijah Cole, and Arvi Gjoka. 2020. The iWildCam 2020 Competition Dataset. *arXiv preprint arXiv:2004.10340* (2020).

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1946–1969.

Sahely Bhadra and Matthias Hein. 2015. Correction of noisy labels via mutual consistency check. *Neurocomputing* 160 (2015), 34–52.

Daniel Bolya, Sean Foley, James Hays, and Judy Hoffman. 2020. Tide: A general toolbox for identifying object detection errors. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*. Springer, 558–573.

Gabriel Cadamuro, Ran Gilad-Bachrach, and Xiaojin Zhu. 2016. Debugging machine learning models. In *ICML Workshop on Reliable Machine Learning in the Wild*, Vol. 103.

Harrison Chase. 2022. *LangChain*. https://github.com/langchain-ai/langchain

Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. *ACM SIGPLAN Notices* 48, 6 (2013), 3–14.

Rada Chirkova and Jun Yang. 2012. *Materialized Views*. Now Publishers Inc., Hanover, MA, USA.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168* (2021).

Edgar F Codd. 1970. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377–387.

Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. 2016. The cityscapes dataset for semantic urban scene understanding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 3213–3223.

Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

Wenjie Du, David Côté, and Yan Liu. 2023. Saits: Self-attention-based imputation for time series. *Expert Systems with Applications* 219 (2023), 119619.

Aarohi Srivastava et al. 2023. Beyond the Imitation Game: Quantifying and extrapolating the capabilities of language models. *Transactions on Machine Learning Research* (2023). https://openreview.net/forum?id=uyTL5Bvosj

Daniel J Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L Sangiovanni-Vincentelli, and Sanjit A Seshia. 2019. Scenic: a language for scenario specification and scene generation. In *Proceedings of the 40th ACM SIGPLAN*

Conference on Programming Language Design and Implementation. 63–78.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2 ed.). Prentice Hall Press, USA.

Parke Godfrey, John Grant, Jarek Gryz, and Jack Minker. 1998. Integrity constraints: Semantics and applications. In *Logics for databases and information systems*. Springer.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

Shreya Havaldar, Bhumika Singhal, Sunny Rai, Langchen Liu, Sharath Chandra Guntuku, and Lyle Ungar. 2023. Multilingual Language Models are not Multicultural: A Case Study in Emotion. In *Proceedings of the 13th Workshop on Computational Approaches to Subjectivity, Sentiment, & Social Media Analysis*. 202–214.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

I Herman. 2013. Eleven sparql 1.1 specifications are w3c recommendation. *w3. org* (2013).

Matthew Honnibal and Ines Montani. 2017. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. (2017). To appear.

Andrew Ilyas, Sung Min Park, Logan Engstrom, Guillaume Leclerc, and Aleksander Madry. 2022. Datamodels: Predicting predictions from training data. *arXiv preprint arXiv:2202.00622* (2022).

Jitesh Jain, Jiachen Li, MangTik Chiu, Ali Hassani, Nikita Orlov, and Humphrey Shi. 2022. OneFormer: One Transformer to Rule Universal Image Segmentation. *arXiv* (2022).

Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems* 1 (2019), 27–39.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. 2017. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 463–478.

Daniel Kang, Deepti Raghavan, Peter Bailis, and Matei Zaharia. 2018. Model assertions for debugging machine learning. In *NeurIPS MLSys Workshop*, Vol. 3. 10.

Pekka Kilpeläinen. 2012. Using XQuery for problem solving. *Software: Practice and Experience* 42, 12 (2012), 1433–1465.

Edward Kim, Divya Gopinath, Corina Pasareanu, and Sanjit A Seshia. 2020. A programmatic and semantic approach to explaining and debugging neural network based object detectors. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 11128–11137.

Pang Wei Koh and Percy Liang. 2017. Understanding black-box predictions via influence functions. In *International conference on machine learning*. PMLR, 1885–1894.

Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of explanatory debugging to personalize interactive machine learning. In *Proceedings of the 20th international conference on intelligent user interfaces*. 126–137.

Keita Kurita, Nidhi Vyas, Ayush Pareek, Alan W Black, and Yulia Tsvetkov. 2019. Measuring Bias in Contextualized Word Representations. In *Proceedings of the First Workshop on Gender Bias in Natural Language Processing*. 166–172.

Chonho Lee, Zhaojing Luo, Kee Yuan Ngiam, Meihui Zhang, Kaiping Zheng, Gang Chen, Beng Chin Ooi, and Wei Luen James Yip. 2017. Big healthcare data analytics: Challenges and applications. *Handbook of large-scale distributed computing in smart healthcare* (2017), 11–41.

Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data Knowl. Eng.* 69, 1 (jan 2010), 3–28. https://doi.org/10.1016/j.datak.2009.08.006

Tao Li, Daniel Khashabi, Tushar Khot, Ashish Sabharwal, and Vivek Srikumar. 2020. UNQOVERing Stereotyping Biases via Underspecified Questions. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. 3475–3489.

Paul Pu Liang, Chiyu Wu, Louis-Philippe Morency, and Ruslan Salakhutdinov. 2021. Towards understanding and mitigating social biases in language models. In *International Conference on Machine Learning*. PMLR, 6565–6576.

Changliu Liu, Tomer Arnon, Christopher Lazarus, Christopher Strong, Clark Barrett, and Mykel J Kochenderfer. 2021. Algorithms for Verifying Deep Neural Networks. *Foundations and Trends® in Optimization* 4, 3-4 (2021), 244–404.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379* (2023).

Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated transpilation of imperative to functional code using neural-guided program synthesis. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022), 1–27.

Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*. PMLR, 2430–2439.

Aaditya Naik, Adam Stein, Yinjun Wu, Mayur Naik, and Eric Wong. 2024. *TorchQL: A Programming Framework for Integrity Constraints in Machine Learning*. https://doi.org/10.5281/zenodo.10723160

Aaditya Naik, Yinjun Wu, Mayur Naik, and Eric Wong. 2023. Do Machine Learning Models Learn Common Sense? *arXiv preprint arXiv:2303.01433* (2023).

Nikita Nangia, Clara Vania, Rasika Bhalerao, and Samuel Bowman. 2020. CrowS-Pairs: A Challenge Dataset for Measuring Social Biases in Masked Language Models. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1953–1967.

The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data* (San Diego, California, USA) *(SIGMOD '92)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/130283.130294

Geoff Pleiss, Danlu Chen, Gao Huang, Tongcheng Li, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Memory-efficient implementation of densenets. *arXiv preprint arXiv:1707.06990* (2017).

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. 2016. Big data analytics on Apache Spark. *International Journal of Data Science and Analytics* 1 (2016), 145–164.

Hadi Salman, Saachi Jain, Andrew Ilyas, Logan Engstrom, Eric Wong, and Aleksander Madry. 2022. When does Bias Transfer in Transfer Learning? *arXiv preprint arXiv:2207.02842* (2022).

Sebastian Schelter, Felix Biessmann, Dustin Lange, Tammo Rukat, Phillipp Schmidt, Stephan Seufert, Pierre Brunelle, and Andrey Taptunov. 2019. Unit testing data with deequ. In *Proceedings of the 2019 International Conference on Management of Data*. 1993–1996.

Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. 2012. Predicting in-hospital mortality of icu patients: The physionet/computing in cardiology challenge 2012. In *2012 Computing in Cardiology*. IEEE, 245–248.

Sahil Singla and Soheil Feizi. 2021. Salient ImageNet: How to discover spurious features in Deep Learning? *arXiv preprint arXiv:2110.04301* (2021).

Sahil Singla, Besmira Nushi, Shital Shah, Ece Kamar, and Eric Horvitz. 2021. Understanding failures of deep networks via robust feature extraction. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 12853–12862.

Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. *Acm Sigplan Notices* 51, 6 (2016), 326–340.

Eric Michael Smith, Melissa Hall, Melanie Kambadur, Eleonora Presani, and Adina Williams. 2022. "I'm sorry to hear that": Finding New Biases in Language Models with a Holistic Descriptor Dataset. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*. 9180–9211.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.

The NumPy Team. 2024. *NumPy*. https://numpy.org/

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 303–314. https://doi.org/10.1145/3180155.3180220

John W Tukey et al. 1977. *Exploratory data analysis*. Vol. 2. Reading, MA.

Daisuke Wakabayashi. 2018. Self-driving uber car kills pedestrian in Arizona, where Robots Roam. https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html

Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 41–53.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

Benjamin Wilson, Judy Hoffman, and Jamie Morgenstern. 2019. Predictive inequity in object detection. *arXiv preprint arXiv:1902.11097* (2019).

Eric Wong, Shibani Santurkar, and Aleksander Madry. 2021. Leveraging sparse linear layers for debuggable deep networks. In *International Conference on Machine Learning*. PMLR, 11205–11216.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*.

Cheng You, Dennis KJ Lin, and S Stanley Young. 2018. Time series smoother for effect detection. *PloS one* 13, 4 (2018), e0195360.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*.

John R Zech, Marcus A Badgeley, Manway Liu, Anthony B Costa, Joseph J Titano, and Eric Karl Oermann. 2018. Variable generalization performance of a deep learning model to detect pneumonia in chest radiographs: a cross-sectional study. *PLoS medicine* 15, 11 (2018), e1002683.

Jieyu Zhao, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. 2017. Men Also Like Shopping: Reducing Gender Bias Amplification using Corpus-level Constraints. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. 2979–2989.

Received 21-OCT-2023; accepted 2024-02-24