Cloudy with High Chance of DBMS: A 10-year Prediction for Enterprise-Grade ML

Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Gowdal, Matteo Interlandi, Alekh Jindal, Konstantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, Yiwen Zhu

name.surname@microsoft.com
Microsoft

ABSTRACT

Machine learning (ML) has proven itself in high-value web applications such as search ranking and is emerging as a powerful tool in a much broader range of enterprise scenarios including voice recognition and conversational understanding for customer support, autotuning for videoconferencing, intelligent feedback loops in large-scale sysops, manufacturing and autonomous vehicle management, complex financial predictions, just to name a few.

Meanwhile, as the value of data is increasingly recognized and monetized, concerns about securing valuable data and risks to individual privacy have been growing. Consequently, rigorous data management has emerged as a key requirement in enterprise settings.

How will these trends (ML growing popularity, and stricter data governance) intersect? What are the unmet requirements for applying ML in enterprise settings? What are the technical challenges for the DB community to solve? In this paper, we present our vision of how ML and database systems are likely to come together, and early steps we take towards making this vision a reality.

1. INTRODUCTION

Machine learning (ML) has proven itself in high-value consumer applications such as search ranking, recommender systems and spam detection [45, 19]. These applications are built and operated by large teams of experts, and run on massive dedicated infrastructures.[1] The (exorbitant) human and hardware costs are well justified by multi-billion dollar paydays. This approach is entirely impractical when it comes to the (ongoing) mainstream adoption of ML.

Enterprises in every industry are developing strategies for digitally transforming their business at every level. The core idea is to continuously monitor all aspects of the business, actively interpret the observations using advanced data analysis—including ML—and integrate the learnings into appropriate actions that improve business outcomes. We predict that in the next 10 years, hundreds of thousands of small teams will build millions[2] of ML-infused applications—most just moderately remunerative, but with huge collective value.

[1] ML.NET [20] alone took dozens of engineers over a decade.

10th Annual Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020, Amsterdam, Netherlands.

When it comes to leveraging ML in *enterprise* applications, especially in regulated environments, the level of scrutiny for data handling, model fairness, user privacy, and debuggability will be substantially higher than in the first wave of ML applications. Consider the healthcare domain: ML models may be trained on sensitive medical data, and make predictions that determine patient treatments—copying CSV files on a laptop and maximizing *average* model accuracy just doesn't cut it! We refer to this new class of applications as *Enterprise Grade Machine Learning (EGML).*

In this paper, we speculate on how ML and database systems will evolve to support EGML over the next several years. Database management systems (DBMSs) are the repositories for high-value data that demands security, fine-grained access control, auditing, high-availability, etc. Over the last 30 years, whenever a new data-related technology has gained sufficient adoption, inevitably DBMS vendors have sought to absorb the technology into their mainstream products. Examples include Object-Oriented [21], XML [31], and Big Data [16] technologies. Indeed, ML is no exception if we consider SQL Server Analysis Services [2], SQL Server R and Python integration [4], and Big Query support for ML [6]. *Is the future then that ML will be assimilated by the DBMS?*

We believe that this is too simplistic, and understanding the path forward requires a more careful look at the various aspects of EGML, which we divide into three main categories: *model development/training*, *model scoring* and *model management/governance*.

Train in the Cloud. First, we are witnessing an ongoing revolution in frameworks for training an increasingly broad range of ML model classes. Their very foundations are still undergoing rapid development [26]. Often, these developments happen in conjunction with innovations in hardware. The rapidly expanding community of data scientists who train models are developing sophisticated environments for managing and supporting the iterative process of data exploration, feature engineering, model training, model selection, model deployment, etc.—prominent examples include [43, 11]. These large, complex, evolving infrastructures are a good fit with managed cloud service infrastructure. Moreover, model training requires centralized data, is characterized by spiky resource usage, and benefits from access to the latest hardware. This leads us to believe that model training and development will happen in either private or public clouds.

[2] From our analysis on >4M Python notebooks from public GitHub repositories [48], we conservatively estimate that 10% of the world's developers will use ML in the next 10 years—totalling 20M engineering years.

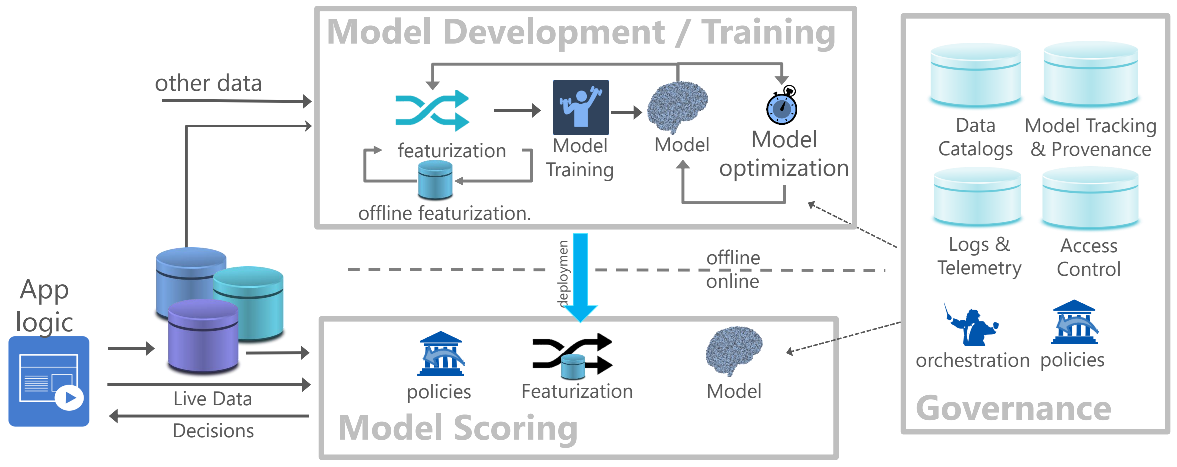


Figure 1: Flock reference architecture for a canonical data science lifecycle.

Score in the DBMS. Second, while the models may be centrally trained the resulting inference pipelines will be deployed everywhere: in the cloud, on-prem, and on edge devices to make inferences ("scoring") where the data is. This raises the question of whether doing inference on data stored in a DBMS can be done as an extension of the query runtime, without the need to exfiltrate the data. We strongly believe this can and must be supported. It appears likely that the most widely studied or promising families of models can be uniformly represented [43, 47]. Given a particular model we can express how to score it (i.e., perform inference) on an input using an appropriate algebra, and compile these algebraic structures into highly optimized code for different execution environments and hardware [25]. Taken together, these observations suggest that we need to consider how to incorporate ML scoring as a foundational extension of relational algebra, and an integral part of SQL query optimizers and runtimes—we present a concrete proposal in § 4.1.

Governance everywhere. Third, we believe that all data, including deployed models—models are, in fact, best thought of as derived data—and the inferences made using them, will need to be robustly governed. The deployment of ML models and their use in decision making via inference leads to many significant challenges in governance. For example, regulations such as GDPR [32] and concerns such as model bias and explainability motivate tracking provenance all the way from data used for training through to decisions based on scoring of trained models. In turn, this requires efficient support for versioning data. While the ML community is focused on improvements in algorithms and training infrastructures, we see massive need for the DB community to step up in the areas of secure data access, version management, and provenance tracking and governance—we discuss initial work in this area in § 4.2.

In summary, the future is likely *cloudy with a high chance of DBMS, and governance throughout.* We describe how our vision is shaped by customer conversations, data and market analysis, and our direct experience as well as present several open problems. We conclude by highlighting promising initial results from a few of the solutions that we are working on.

2. THE FLOCK VISION

In this section, we present our vision for *Flock*, a reference architecture to support the canonical data science lifecycle for EGML applications. *Flock* is our vehicle to explore assumptions (§ 3), discover open problems and validate initial solutions (§ 4). We start from a key observation: *Machine Learning models are software artifacts derived from data.* The resulting dual nature of software artifacts and derived data provides us with a useful lens to understand the role of the DB community in the EGML revolution.

The lifecycle shown in Figure 1 begins with a (typically) offline phase, where a data scientist (and more and more frequently any software engineer) gathers data from multiple data sources, transforms them via several featurization steps and models reality using learning algorithms. Today, this phase is very manual and sadly closer to a black art than an engineering discipline. Looking at ML as software, we expect the ML and Software Engineering communities to provide us with automation [35], tooling, and engineering best practices—ML will become an integral part of the DevOps lifecycle. Looking at ML models as derived data, the DB community must address data discovery, access control and data sharing, curation, validation, versioning and provenance (§ 4.2). Moreover, today's prevalent abstraction for data science is imperative Python code orchestrating data-intensive processing steps, each performed within a native library. This suggests that end-to-end ML pipelines can be approached as inherently optimizable dataflows (§ 4.1).

The second stage in the lifecycle is entered when a model is selected and is ready to be deployed. Using the models-as-software lens, deployment consists of packaging the entire inference pipeline (model + all data preprocessing steps) in a way that preserves the exact behavior crafted by the data scientist in the training environment. Next, deployment envolves finding a suitable hosting infrastructure for scoring the model. Today's best practice is to package models in costly containers and hope that enough of the environment is preserved to ensure correctness[3]. Recall that in EGML settings individual decisions could be very consequential (e.g., loan acceptance, or choice of medical treatment), so "average model accuracy" is not a sufficient validation metric. Switching to our models-as-data lens, we observe that they must be subject to GDPR-style scrutiny, and their storage and querying/scoring must be secured and auditably tracked—e.g., [24] has shown how to extract credit card numbers memoized in a trained neural network language model. Also, privacy and fairness implications must be handled carefully. Moreover, as the underlying data evolves, models need to be updated. To retain

[3]This is optimistic (e.g., is floating point precision guaranteed when running a container across Linux/Windows, x64/ARM?)

consistency for complex applications multiple models might have to be updated transactionally. DBMSs have long provided these type of enterprise features for operational data, and we propose to extend them to support model scoring. While this was our primary motivation, our early experiments suggest that in-DB model scoring actually allows us to deliver $5\times$ to $24\times$ speedups over standalone state of the art solutions!

Model predictions usually come in the form of single numbers or vectors of numbers (e.g., the probability of each class in a classification problem). To act on a prediction, it must be transformed to domain terms (e.g., the name of the winning class). But actions are typically more nuanced and involve policies that encode business constraints and might actually override a model's prediction under certain circumstances. Systematizing this policy space is important, as we have discussed in [30].

Beside basic orchestration (of what happens when) in project lifecycle, management and governance for data and models throughout is vital. Access to a deployed model must be controlled, similar to how access to data or a view is controlled in a DBMS. Provenance here plays a key role and has two distinct applications—looking at models as software artifacts, we must be able to verify them or debug them, even as they evolve due to re-training. From a model-as-data viewpoint, we must be able to determine how a model was derived and from which snapshot of (training) data, in order to interpret the predictions and answer questions such as whether and why certain decisions were biased. This leads to the need for pervasive and automated tracking of provenance from training through deployment to scoring (§ 4.2).

Given this context, we argue that: 1) the *ML development/training will happen in the Cloud*; 2) *Models must be stored and scored in managed environments such as a DBMS*; and 3) *Provenance needs to be collected across all phases*.

3. THE VANTAGE POINT

Our perspective on *what Enterprise-grade ML (EGML) will look like in 10 years* is shaped by multiple inputs.

First-hand experience. Collectively, the authors of this paper have extensive experience in using ML technologies in production settings, e.g., content recommenders [19], spam filters [45], big data learning optimizers [51, 54, 37, 53], ML-based performance debuggers, Azure cloud optimizations based on customer load predictions, self-tuning streaming systems, and auto-tuning infrastructures for SQL Server internals. Many of us have also been working on systems for ML technologies, including big data infrastructure [29, 10, 36], ML toolkits [20, 14, 15], and the systems that orchestrate it all in the cloud [11].

This experience has led to one key insight: **"An ML model is software derived from data"**. This means that ML presents characteristics that are typical of software (e.g., it requires rich and new CI/CD pipelines), and of data (e.g., the need to track lineage)—hence, the database community is well positioned to play a key role in EGML, but much work is needed. We discuss some of the problems we tackle in § 4.

A second—and painfully clear—observation is that **the actual model development represents less than 20% of most data science project lifecycle** [5]. The majority of the time is spent on collecting and cleaning the data, with the reminder on operationalizing the best model.

Conversations with enterprises. We have engaged with many large, sophisticated enterprises, including: (i) a financial institution seeking to streamline its loan approval process, (ii) a marketing firm identifying which customers to target for promotions,

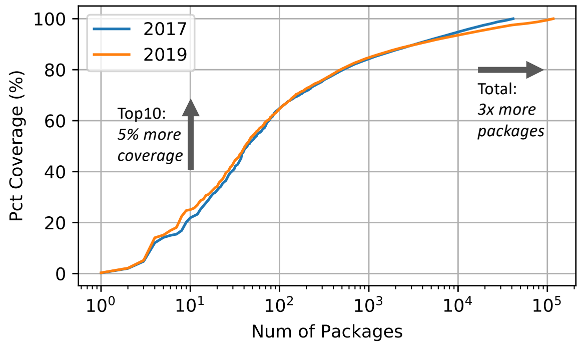


Figure 2: Notebook coverage (%) for top-K packages.

(iii) a sports company predicting athletes' performance, (iv) a health insurance agency aiming to predict patient recidivism, and (v) a large automotive company modeling recalls, customer satisfaction, and marketing.

A key learning from these conversations is that compared to "unicorn" ML applications like web search, these enterprise applications are characterized by smaller teams with domain expertise rather than deep algorithmic or systems expertise. On the other hand, their platform requirements are much more stringent around auditing, security, privacy, fairness, and bias.[4] This is particularly true for regulated industries. Existing ML technologies are not ready to support these applications in a safe, cost-effective manner. We also learned that is increasingly common for a data science project to produce a large number of localized/specialized models, rather than a single uber model—e.g., a scam-detection model per ATM vs a single global model. This creates novel challenges on model management, deployment, tracking, (transactional) rolling forward/backward among model versions etc. In short: **Typical applications of ML are built by smaller, less experienced teams, yet they have more stringent demands**.

GitHub analysis. To get a feel for trends in the broader data science community, we *downloaded and analyzed* >4 *million public Python notebooks from GitHub*, plus hundreds of thousands of data science pipelines from within Microsoft [48]. Over 70% of the notebooks focused on ML. Moreover, we analyzed hundreds of versions for popular Python packages. The details of this analysis are beyond the scope of this paper, but we make a few key observations. Figure 2 shows the fraction of notebooks that would be completely supported, if we only covered the K most popular packages (for varying values of K). The shift between 2017 and 2019 suggests that the field is still expanding quickly (many more packages) but also that we are seeing an initial convergence (a few packages are becoming dominant). For example, `numpy`, `pandas` and `sklearn` are solidifying their position. We also observe very limited adoption of solutions for testing/CI-CD/model tracking (MLFlow [43] is still not very popular despite its relevance to EGML).

Overall this suggests that **systems aiming to support EGML must provide broad coverage, but can focus on optimizing a core set of ML packages.**

Competitive landscape. Many of the companies that built the first "unicorn" ML applications also developed systems to support the data science lifecycle. Some of those tech stacks were later released as open source while others lead to services on the public

[4]This is not intended to suggest that unicorn applications do not share these requirements; rather, enterprise teams want off-the-shelf platforms that have a much higher level of support built-in, whereas unicorn teams have typically built everything from scratch.

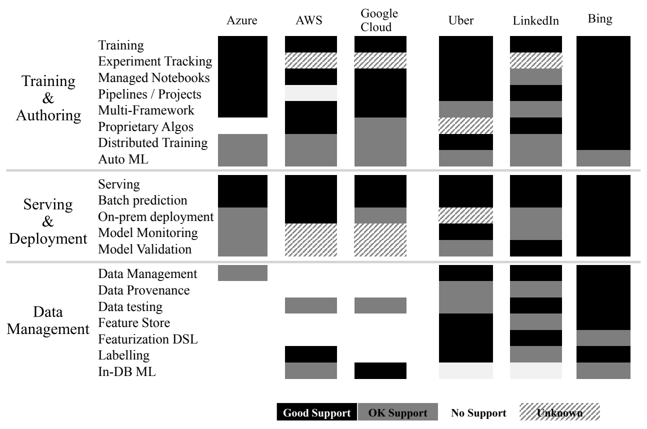


Figure 3: ML Systems in the public cloud and major companies.

cloud. In Figure 3, we compare some of the most mature systems in this area (with accessible information), based on the level to which they support different features[5]. Note that the area is dynamic and that the table reflects our understanding of these systems at the time of writing. We consider Bing, Uber [7] and LinkedIn's ProML [8] as examples of proprietary infrastructures powering "unicorn" applications. Also, all major public cloud providers have services to support enterprise machine learning [11, 12, 9].

Analyzing those systems, we identified key feature areas: Training, Deployment and Data Management. A detailed discussion is beyond the scope of this submission. But we observe two major trends: 1) **mature proprietary solutions have stronger support for data management**—this is consistent with our own direct experience, and 2) **providing complete and usable third-party solutions in this space is non-trivial**—or the cloud vendors who already had internal versions of this would have already done so. We speculate that this relates to the extra challenges introduced by EGML, and believe this area is primed for disruptive research, as we discuss in more details in § 4.

Research in the ML community. Research in the area of machine learning systems is plentiful, and too large an area to do justice to here. Instead, we focus on major trends that we observed over the last 15 years that influenced our thinking on the intersection of machine learning and data management systems.

After an initial focus on algorithms, the ML community has concentrated (in rough order of appearance) on: 1) systems for training, 2) systems for scoring, 3) AutoML solutions, and 4) responsible AI. The systems for training area was initially dominated by big-data extensions [3, 44] first and HPC-based solutions [50], and later by parameter servers with bounded staleness [28]. More recent attempts such as ML.NET [20] and TFX [23] have borrowed more profoundly from the dataflow/database literature to build ML-native solutions. Systems for scoring were vastly ignored until systems such as Clipper [27], Pretzel [42] and TensorFlow serving [46] came to fruition. They draw heavily from streaming systems and optimizing compilers in their design. As ML adoption began to broaden, AutoML solutions began to appear [35]. Lately, interest in bias, fairness and responsible use of machine learning is exploding, though only limited solutions exist. This aligns with feedback from enterprise customers (i.e., "automate it, and don't get me sued").

[5]This is ostensibly a subjective judgement based on a few weeks of analysis of marketing material, code skimming, and light experimentation.

We conclude that **data platforms play a key role to achieve fast and reliable training and scoring, and that explicit metadata management and provenance tracking are foundational for responsible AI and AutoML solutions**.

4. OPEN PROBLEMS & ADVANCES

The vision for EGML we presented is an exciting one and presents many challenging problems. We summarize some key challenges below and present some of our ongoing work. We focus on two categories that require attention from the DB community and are not well understood: 1) the systems support required to go from a trained model to decisions, and 2) data management for ML.

4.1 From Model to Decision: Inference

Much attention has been given to learning algorithms and efficient model training, but models only have value insofar as they are used for inference, to create insights and make decisions. This typically involves a complicated setup of containers for deploying the trained model (as executable code), with applications invoking them via HTTP/REST calls. Further, the containerized code often extends model inference with the implementation of complex application-level policies.

While this containerized approach offers a desirable decomposition of the problem between models and the applications using them, it has significant drawbacks: (1) Many applications use more than one model, with each model applied to the outcome of some (potentially different) data processing step. These assemblies of models and preprocessing steps should be updated atomically. (2) It seems unlikely that this solution will fit the scenarios emerging from the millions of applications we expect in this space (e.g., latency-sensitive decisions and large batch predictions are poorly served). (3) Mixing application-level policies and inference logic makes it hard to separate and measure the impact of the two.

We believe that models should be represented as first-class data types in a DBMS. This will naturally address (1) by allowing database transactions to be used for updating multiple deployed models. To address (2), we believe inference/scoring should be viewed as an extension of relational query processing, and argue for moving model inference close to the data and performing it in-DBMS, without external calls for common types of models. Naturally, this calls for a separation of inference from application-level logic. We present a clean framework for (3) after we briefly summarize our early results on in-DBMS inference. A more in-depth discussion on inference appears in [40].

In-DBMS inference. While in-DBMS inference appears desirable, a key question arises: *Can in-DBMS model inference perform as well as standalone dedicated solutions?*

To this end, several recent works [33, 41, 34, 18] in the database community explore how linear and relational algebra can be co-optimized. To carry this investigation further, we integrated ONNX Runtime [15] within SQL Server and developed an in-database cross-optimizer between SQL and ML to enable optimizations across hybrid relational and ML expressions [40]. Further, we observe that practical end-to-end prediction pipelines are composed of a larger variety of operators (e.g., featurizers such as text encoding and models such as decision trees) often assembled in Python. We leverage static analysis to derive an intermediate representation (IR) amenable to optimization. The list of optimizations we have been exploring is therefore more comprehensive than prior work, and includes the following:

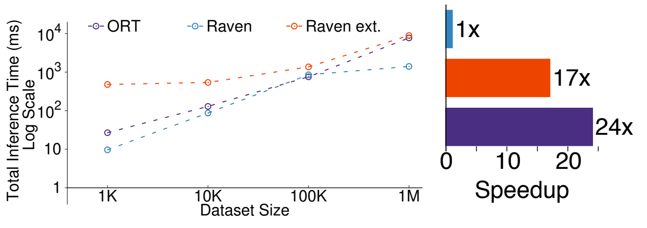


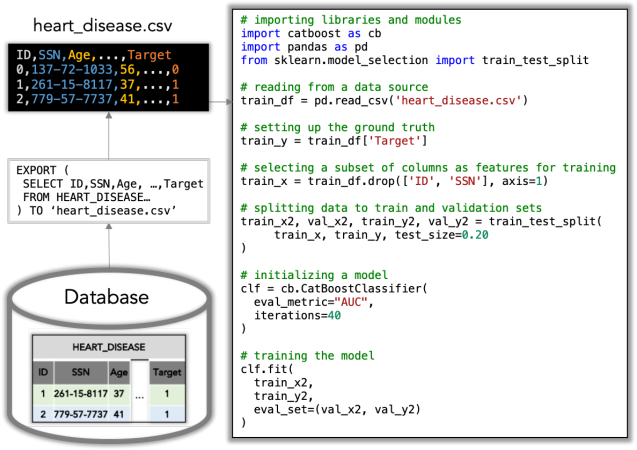
Figure 4: Benefits of integrating ONNX Runtime in SQL Server (left) and of cross-optimizations (right—red bar shows speedup due to model inlining, blue bar also includes predicate-based model pruning).

- predicate-based model pruning, which uses selections from the SQL subsection of the query to simplify the ML subsection;
- model-projection pushdown, which automatically prunes (projects out) unused input feature-columns exploiting model-sparsity;
- model clustering that compiles simplified models for different parts of the input based on data statistics;
- model inlining, which transforms ML operators to relational ones (similar in spirit to [49]);
- physical operator selection based on statistics, available runtime (SQL/ONNX/Python UDFs [52]) and hardware (CPU, GPU).

In Figure 4 we present two key results: (1) performance of integrating ONNX Runtime (ORT) within SQL Server, both through in-process execution (Raven) and out-of-process (Raven ext.); and (2) benefit of cross-optimizations, including model inlining and predicate-based model pruning. The results show that SQL Server integration provides up to 5.5× over standalone ORT (due to automatic parallelization of the inference task in SQL Server) and up to 24× from our combined optimizations. Early results indicate that in-DBMS inference is very promising.

Bridging the model-application divide. ML applications need to transform the model predictions into actionable decisions in the application domain. However, the mathematical output of the model is rarely the only parameter considered before a decision is made. In real deployment scenarios, business rules and constraints are important factors that need to be taken into account before any action is taken. As a concrete example, we have built models to automate the selection of parallelism for large big data jobs to avoid resource wastage (in the context of Cosmos clusters [38]). While models are generally accurate, they occasionally predict resource requirements in excess of the amounts allowed by user-specified limits. Business rules expressed as policies then override the model.

Obviously, business rules and requirements can vary between different applications and environments. To that end, we employ a generic and extensible module [30] that takes as input user-defined *policies* which introduce various business constraints on top of EGML workloads. The module continuously monitors the output of the ML models and applies the specified policies before taking any further action in the application domain. It also maintains the system state and actions taken over time allowing us to easily debug and explain the system's actions. Finally, it makes sure that the actions happen in a transactional way, rolling back in case of failures when needed. Overall this closes the loop between model and application,



```
ID,SSN,Age,...,Target
0,137-72-1033,56,...,0
1,261-15-8117,37,...,1
2,779-57-7737,41,...,1
```

```
EXPORT (
 SELECT ID,SSN,Age, …,Target
 FROM HEART_DISEASE…
) TO ‘heart_disease.csv’
```

HEART_DISEASE

ID	SSN	Age		Target
1	261-15-8117	37	...	1
2	779-57-7737	41		1

```
# importing libraries and modules
import catboost as cb
import pandas as pd
from sklearn.model_selection import train_test_split

# reading from a data source
train_df = pd.read_csv('heart_disease.csv')

# setting up the ground truth
train_y = train_df['Target']

# selecting a subset of columns as features for training
train_x = train_df.drop(['ID', 'SSN'], axis=1)

# splitting data to train and validation sets
train_x2, val_x2, train_y2, val_y2 = train_test_split(
     train_x, train_y, test_size=0.20
)

# initializing a model
clf = cb.CatBoostClassifier(
  eval_metric="AUC",
  iterations=40
)

# training the model
clf.fit(
  train_x2,
  train_y2,
  eval_set=(val_x2, val_y2)
)
```

Figure 5: Data science pipeline

providing us visibility necessary for both debugging and end-to-end accountability.

Next, we discuss the requirements for managing data for ML.

4.2 Data Management for ML

Data Discovery, Access and Versioning. One of the main challenges ML practitioners face today revolves around data access and discovery. Training data commonly contains tabular data, but also images, video or other sensor data. This gives rise to a predominantly file-based workflow. Only a small fraction of the >4 million notebooks we analyzed makes use of a database access library. This is surprising, as the vast majority of the pipelines ultimately use Pandas [17], a structured DataFrame, to interact with this data. This state-of-the-art is deeply unsatisfying: *Data Discovery support is virtually non-existent* . This is especially troubling as data augmentation is one of the best strategies to improve a model.

Worse, *data versioning is largely unsolved in this paradigm:* A model is the result of both its training code and the training data. Both need to be versioned. And file versioning technologies fail to address key needs of data versions; they often can only represent a deletion via a history rewrite. More fundamentally, files are not the atomic unit of training data: an individual data point may be stored in a file, but equally likely, many files represent one data point, or one file contains many data points.

Hence, we believe that **there is an open need for queryable data abstractions, lineage-tracking and storage technology that can cover heterogenous, versioned, and durable data.**

*Model Management.*We have argued that ML models are software artifacts created from data, and must be secured, tracked and managed on par with other high-value data. DBMSs provide a convenient starting point thanks to their support for enterprise-grade features such as security, auditability, versioning, high availability, etc. To be clear, we are not suggesting that all data management needs to be inside a relational DBMS; indeed, we see a trend towards comprehensive data management suites that span all of a user's data across one or more repositories. Our point is that managing models should be treated on par with how high-value data is managed, whether in a DBMS (the most widely available option currently) or in emerging cross-repository managed environments.

Model Tracking and Provenance. Models are software of consequence. Their genesis needs to be tracked. To achieve that, the full *provenance* of a model must be known for debugging/auditing.

We need to capture not only the code that trained the model, but also the (training) data that went into it, together with its full, tamper-proof lineage. There are multiple industry efforts to capture the inner training loop of this lineage [43, 13]. This must be expanded to the full lineage, and also automated to achieve the scale we expect.

Figure 5 shows an example of a data science pipeline. The data scientist first selects a subset of patient-related data stored in a relational database and saves them in a CSV file. This file can then be accessed by a Python script to train a machine learning model. This pipeline involves different tools and languages (SQL, Python) and data formats (SQL tables, flat files).

Tracking provenance across such pipelines can enable various important applications such as model linting, compliance testing, and impact analysis. In this particular example, the label used to train the model (TARGET column) is also part of the feature set. Automatically analyzing the Python script to identify the provenance relationships between the columns of the dataset and the trained model can help identify such mistakes. Similar techniques can also be used to detect compliance violations such as usage of PII information by a machine learning model. In our example, the dataset contains sensitive patient data that include the SSN and ID columns. However, these columns are dropped from the Python script before training the machine learning model, and as a result the pipeline is compliant with the regulations. Finally, automatic tracking of provenance can help identify the set of applications that are affected by changes in the original data sources. For example, one can use such information to detect which machine learning models will be impacted when a column of the dataset in the database is dropped because it contains corrupted data.

Enabling such applications by automatically capturing provenance information across such pipelines is challenging:

- **C1. Provenance data model.** Data elements in EGML workloads are polymorphic (e.g., tables, columns, rows, ML models, and hyper-parameters) with inherent temporal dimensions (e.g., a model may have multiple versions, one for each re-run of a training pipeline). As such, and in contrast to traditional data models of provenance over DBMSs, EGML workloads dictate polymorphic and temporal provenance data models. Such data models are hard to design, capture, maintain, and query.

- **C2. Provenance capture.** EGML workloads typically span multiple systems and runtimes (e.g., a Python script may fetch data from multiple databases to train a model). These systems might have different architecture and programming constructs (e.g., declarative vs. imperative interfaces). Extracting a meaningful provenance data model in this setting requires different capture techniques tailored specifically to each system and runtime.

- **C3. Provenance across disparate systems.** Even if we capture provenance on top of each system and runtime in isolation, we still require to combine this information across systems (e.g., if we change a column in a database, models trained in Python that depend on this column may need to be invalidated and retrained). Hence, EGML workloads require protocols for consolidating and communicating the provenance information across systems.

Our initial solution. Our solution consists of three major modules: the *SQL Provenance module*, the *Python Provenance module* and the *Catalog*. The Catalog (we use Apache Atlas [22]) stores all the provenance information and acts as the bridge between the SQL and the Python Provenance modules. It allows us to capture end-to-end provenance across different systems—hence, provides a principal way to address **C3**. Next, we present the high-level architecture and some preliminary experimental results (full papers with the detailed description of the system design and associated algorithms behind the SQL and Python Provenance modules are under preparation).

Provenance in SQL. Our SQL provenance module currently focuses on capturing coarse-grained provenance under two modes, traditionally referred to as eager and lazy. Under eager provenance capture, given a query, the module parses it to extract coarse-grained provenance information (i.e., input tables and columns that affected the output, with connections modelled as a graph). Under lazy provenance capture, the module gets as input the query log of the database and constructs the provenance data model, only this time by accounting the whole query history. Under both modes, the module populates the Catalog accordingly.

To scale across databases, the parsing module utilizes Apache Calcite [1] that provides parsers and adapters across databases—hence, provides us a way towards addressing **C2**. For cases where Apache Calcite cannot parse queries, we specialize to the parser of the corresponding engine. Furthermore, note that all data stored in the Catalog is versioned (e.g., an `INSERT` to a table results in a new version of the table in the provenance data model)—hence, we address the temporal aspect of **C1**. The table below shows the provenance capture performance (latency and provenance graph size) for queries generated out of all templates in TPC-H and TPC-C:

Dataset	**#Queries**	**Latency**	**Size(nodes+edges)**
TPC-H	2,208	110s	22,330
TPC-C	2,200	124s	34,785

These early findings indicate that a) the per-query capture latency can be significant and b) the provenance data model can become substantially large in size (e.g., a table with as many versions as the number of insertions that have happened to it). For these reasons, we develop optimized capture techniques, through compression and summarization, which are essential towards addressing **C1**.

Provenance in Python. The Python provenance module parses scripts and automatically identifies the lines of code that correspond to feature extraction and model training using a combination of standard static analysis techniques and a knowledge base of ML APIs that we maintain. Through this process, we are able to identify which Python variables correspond to models, hyperparameters, model features and metrics. We can also track the transformations performed on these variables and eventually connect them with the datasets used to generate training data. The Python provenance module accesses the Catalog to collect the output of the SQL provenance module and eventually connect the datasets used in the Python scripts to the columns of one or more DBMS tables.

Dataset	**#Scripts**	**%Models Covered**	**%Training Datasets Covered**
Kaggle	49	95%	61%
Microsoft	37	100%	100%

The above table table shows the coverage currently achieved by the provenance module on the Kaggle dataset [39] and a Microsoft internal dataset of scripts deployed in production. In this experiment, we evaluate how often the module identifies correctly ML models and training datasets in the Python scripts.

5. CONCLUSION AND CALL TO ACTION

We live in interesting times. Database architectures are undergoing major transformations to leverage the elasticity of clouds, and a combination of increased regulatory pressures and data sprawl is forcing us to rethink data governance more broadly. Against this backdrop, the rapid adoption of ML in enterprises raises foundational questions at the intersection of model training, inference and governance, and we believe the DB community needs to play a significant role in shaping the future.

Acknowledgements

We thank Doris Xin, Mohammad Hossein Namaki and Yi Zhang for their contributions in Sections 4.1 and 4.2—full-length papers on each system contribution are ongoing.

6. REFERENCES

[1] Apache Calcite. https://calcite.apache.org.
[2] Microsoft SQL Server Analysis Services. https://docs.microsoft.com/en-us/analysis-services/analysis-services-overview, 2000.
[3] Apache Mahout. https://mahout.apache.org, 2011.
[4] Microsoft SQL Server Machine Learning Services. https://docs.microsoft.com/en-us/sql/advanced-analytics/, 2017.
[5] 2018 Kaggle Machine Learning and Data Science Survey. https://www.kaggle.com/paultimothymooney/2018-kaggle-machine-learning-data-science-survey, 2018.
[6] Google BigQuery ML. https://cloud.google.com/bigquery-ml/docs/bigqueryml-intro, 2018.
[7] Scaling Machine Learning at Uber with Michelangelo. https://eng.uber.com/scaling-michelangelo/, 2018.
[8] Scaling Machine Learning Productivity at LinkedIn. https://engineering.linkedin.com/blog/2019/01/scaling-machine-learning-productivity-at-linkedin, 2018.
[9] Amazon SageMaker. https://aws.amazon.com/sagemaker/, 2019.
[10] Apache Hadoop. https://hadoop.apache.org, 2019.
[11] Azure Machine Learning service. https://azure.microsoft.com/en-us/services/machine-learning-service/, 2019.
[12] Google Cloud AI Platform. https://cloud.google.com/ai-platform/, 2019.
[13] Kubeflow. https://kubeflow.org, 2019.
[14] Microsoft Nimbus ML. https://docs.microsoft.com/en-us/NimbusML, 2019.
[15] Microsoft ONNX Runtime. https://github.com/Microsoft/onnxruntime, 2019.
[16] Microsoft SQL Server 2019. https://www.microsoft.com/en-us/sql-server/sql-server-2019, 2019.
[17] pandas: Python Data Analysis Library. https://pandas.pydata.org/, 2019.
[18] C. R. Aberger, A. Lamb, K. Olukotun, and C. Ré. Levelheaded: A unified engine for business intelligence and linear algebra querying. In *ICDE*, 2018.
[19] D. Agarwal, B. chung Chen, P. Elango, N. Motgi, S. taek Park, R. Ramakrishnan, S. Roy, and J. Zachariah. Online models for content optimization. In *NeurIPS*. 2009.
[20] Z. Ahmed, S. Amizadeh, M. Bilenko, R. Carr, W. Chin, Y. Dekel, X. Dupré, V. Eksarevskiy, E. Erhardt, C. Eseanu, S. Filipi, T. Finley, A. Goswami, M. Hoover, S. Inglis, M. Interlandi, S. Katzenberger, N. Kazmi, G. Krivosheev, P. Luferenko, I. Matantsev, S. Matusevych, S. Moradi, G. Nazirov, J. Ormont, G. Oshri, A. Pagnoni, J. Parmar, P. Roy, S. Shah, M. Z. Siddiqui, M. Weimer, S. Zahirazami, and Y. Zhu. Machine learning at Microsoft with ML.NET. In *KDD*, pages 2448–2458, 2019.
[21] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The object-oriented database system manifesto. In *Deductive and object-oriented databases*. 1990.
[22] Apache Atlas. https://atlas.apache.org/, 2019.
[23] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. TFX: A tensorflow-based production-scale machine learning platform. In *SIGKDD*, 2017.
[24] N. Carlini, C. Liu, Úlfar Erlingsson, J. Kos, and D. Song. The secret sharer: Evaluating and testing unintended memorization in neural networks, 2018.
[25] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.
[26] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud. Neural ordinary differential equations. In *Advances in neural information processing systems*, pages 6571–6583, 2018.
[27] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.
[28] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *Usenix ATC*, 2014.
[29] C. Curino, S. Krishnan, K. Karanasos, S. Rao, G. M. Fumarola, B. Huang, K. Chaliparambil, A. Suresh, Y. Chen, S. Heddaya, R. Burd, S. Sakalanaga, C. Douglas, B. Ramsey, and R. Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *NSDI*, 2019.
[30] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: self-regulating stream processing in heron. *PVLDB*, 2017.
[31] D. Florescu and D. Kossmann. Storing and querying xml data using an rdmbs. *IEEE data engineering bulletin*, 22:3, 1999.
[32] GDPR. https://gdpr-info.eu/, 2016.
[33] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 2012.
[34] D. Hutchison, B. Howe, and D. Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *BeyondMR@SIGMOD*, 2017.
[35] F. Hutter, L. Kotthoff, and J. Vanschoren, editors. *Automated Machine Learning: Methods, Systems, Challenges*. Springer, 2018. In press, available at http://automl.org/book.
[36] V. Jalaparti, C. Douglas, M. Ghosh, A. Agrawal, A. Floratou, S. Kandula, I. Menache, J. S. Naor, and S. Rao. Netco: Cache and i/o management for analytics over disaggregated stores. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 186–198, 2018.
[37] A. Jindal, L. Viswanathan, and K. Karanasos. Query and Resource Optimizations: A Case for Breaking the Wall in Big Data Systems. *CoRR*, abs/1906.06590, 2019.
[38] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan,

J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

[39] Kaggle Dataset. https://www.kaggle.com/datasets, 2019.

[40] K. Karanasos, M. Interlandi, D. Xin, F. Psallidas, R. Sen, K. Park, I. Popivanov, S. Nakandal, S. Krishnan, M. Weimer, Y. Yu, R. Ramakrishnan, and C. Curino. Extending relational query processing with ML inference. In *CIDR*, 2020.

[41] A. Kunft, A. Katsifodimos, S. Schelter, and V. Markl. An intermediate representation for optimizing machine learning pipelines. *PVLDB*, 2019.

[42] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: opening the black box of machine learning prediction serving systems. In *OSDI*, 2018.

[43] MLflow: An open source platform for the machine learning lifecyle. https://mlflow.org, 2019.

[44] Apache Spark MLLib. https://spark.apache.org/docs/latest/ml-guide.html, 2019.

[45] S. Narayan, V. T. Ramarao, B. Tseng, M. Weimer, Y. Maeng, and J.-S. Shue. Anti-spam transient entity classification. US9442881B1, 2011.

[46] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. In *Workshop on ML Systems at NIPS 2017*, 2017.

[47] ONNX: Open Neural Network Exchange Format. https://onnx.ai.

[48] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino, and M. Weimer. Data science through the looking glass and what we found there. *ArXiv e-prints*, 1912.09536, 2019.

[49] K. Ramachandra, K. Park, K. V. Emani, A. Halverson, C. A. Galindo-Legaria, and C. Cunningham. Froid: optimization of imperative programs in a relational database. *PVLDB*, 2017.

[50] A. Sergeev and M. D. Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR*, abs/1802.05799, 2018.

[51] T. Siddiqui, A. Jindal, S. Qiao, H. Patel, and W. Le. Cost Models for Serverless Query Processing: Learning, Retrofitting, and Our Finding. *SIGMOD (to appear)*, 2020.

[52] Execute external scripts in SQL Server. https://tinyurl.com/sqlserver2017, 2019.

[53] L. Viswanathan, A. Jindal, and K. Karanasos. Query and Resource Optimization: Bridging the Gap. In *ICDE*, 2018.

[54] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a Learning Optimizer for Shared Clouds. *PVLDB*, 2018.