Data Platform for Machine Learning

Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal, Sethu Raman, Vishrut Shah, Bochao Shen, Laura Sugden, Kaiyu Zhao, Ming-Chuan Wu*

Apple Inc.
Cupertino, CA

ABSTRACT

In this paper, we present a purpose-built data management system, MLdp, for all machine learning (ML) datasets. ML applications pose some unique requirements different from common conventional data processing applications, including but not limited to: data lineage and provenance tracking, rich data semantics and formats, integration with diverse ML frameworks and access patterns, trial-and-error driven data exploration and evolution, rapid experimentation, reproducibility of the model training, strict compliance and privacy regulations, *etc.* Current ML systems/services, often named MLaaS, to-date focus on the ML algorithms, and offer no integrated data management system. Instead, they require users to bring their own data and to manage their own data on either blob storage or on file systems. The burdens of data management tasks, such as versioning and access control, fall onto the users, and not all compliance features, such as *terms of use*, privacy measures, and auditing, are available.

MLdp offers a minimalist and flexible data model for all varieties of data, strong version management to guarantee re-producibility of ML experiments, and integration with major ML frameworks. MLdp also maintains the data provenance to help users track lineage and dependencies among data versions and models in their ML pipelines. In addition to table-stake features, such as security, availability and scalability, MLdp's internal design choices are strongly influenced by the goal to support rapid ML experiment iterations, which cycle through data discovery, data exploration, feature engineering, model training, model evaluation, and back to data discovery.

The contributions of this paper are: 1) to recognize the needs and to call out the requirements of an ML data platform, 2) to share our experiences in building MLdp by adopting existing database technologies to the new problem as well as by devising new solutions, and 3) to call for actions from our communities on future challenges.

ACM Reference Format:
Pulkit Agrawal, Rajat Arya, Aanchal Bindal, Sandeep Bhatia, Anupriya Gagneja,, Joseph Godlewski, Yucheng Low, Timothy Muss, Mudit Manu Paliwal,, and Sethu Raman, Vishrut Shah, Bochao Shen, Laura Sugden, Kaiyu Zhao, Ming-Chuan Wu. 2019. Data Platform for Machine Learning. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3299869.3314050

1 INTRODUCTION

Machine learning has become increasingly prevalent in the industry today. Across all business sectors, systems and software features have incorporated machine learning, from the data center to in your home, car and pocket. Data is the key ingredient to training successful machine learning models. The data available to train machine learning models has grown tremendously over the last decade, but no systems to support these needs holistically had emerged at the time we sought a solution. In this paper we share the experiences and future challenges of building an integrated data system for machine learning, named MLdp in short for ease of references for the rest of the paper. To understand why such a system is necessary, it is important to understand the challenges machine learning places on data systems.

The rest of the paper is organized as follows. In Section 2, we justify the need of an integrated ML data management system by calling out the requirements and cross-examining existing solutions and systems against those requirements. We discuss the system design in Section 3, which includes the data model, data interface with ML framework integration,

*Primary contact author/email: ming-chuan.wu@apple.com

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06.
https://doi.org/10.1145/3299869.3314050

physical data layout design, and a distributed caching service for MLdp. In Section 4, we highlight the important future extensions as a call-for-action to the community. Finally, we conclude in Section 5.

2 MOTIVATION

In examining the machine learning life cycle, we observe that there are the following distinct stages: data collection, annotation, exploration, feature engineering, experimentation, evaluation, and finally deployment. Variations of these stages have been discussed in [21], [9] and [6]. At every stage of the iterative life cycle, data is cleansed, transformed, and engineered to fit the needs of subsequent ML stages. The key insights in our observation from existing solutions are as follows. The emphasis of existing solutions has been on training, experimentation, evaluation, and deployment phases of the life cycle, as shown in Table 1. However, little emphasis[1] has been made on the needs of an integrated ML data system. Instead, during data collection, data are often injected into cloud blob storage systems. The heavy-lifting ETL (extract-transform-load) pipelines are often performed using data processing systems, such as MapReduce or Spark, with the results stored in distributed file systems or back to the blob storage. The crowd-sourced annotation tasks often require data to be exported and imported to/from publicly accessible data storage systems. The derived and transformed data between different stages are simply stored in storage systems based on the preferences of the data engineers. These data silos are used as simple, passive data stores, leaving the burden of integrating data access with ML tasks to the ML engineers. The ML engineers often face the challenges of coping with different data layouts and formats and of different data access interfaces from different silos throughout different stages instead of focusing on ML challenges.

As illustrated in Figure 1, developers, using different tools in every stage in ML life cycle, need to develop an understanding for data internals, leading to a cascading physical data dependency. This intertwined physical dependence further hinders the ability to offer data provenance, data versioning, compliance, auditing, usage analytics, *etc.* Even worse, it hampers the rate of ML innovations, because every new project needs to re-learn these complex dependencies. Furthermore, in large organizations, we observed various teams solving similar ML problems, often without knowledge of each other. A central data management system will enable these teams to discover and share relevant data and to foster new collaborations between teams.

[1] Only since recent years, the need of an ML data platform has attracted attention from companies and our community, *e.g.*, DEEM (International Workshop on Data Management for End-to-End Machine Learning).

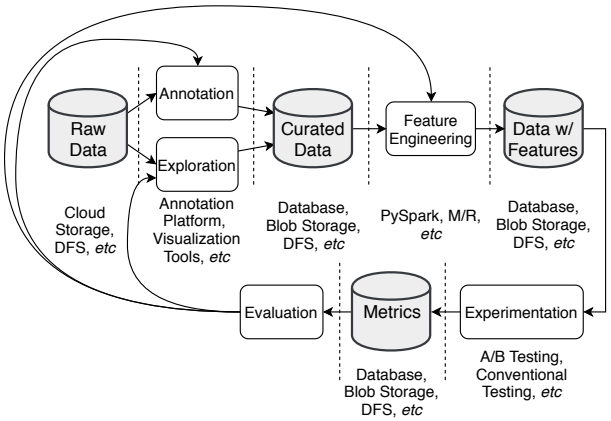


Figure 1: Conventional ML workflow and data silos.

Based on these observations, we see a strong need to build a dataset management system that integrates into the ML life cycle, guarantees compliance, enforces legal terms of use, offers a high-level data access abstraction to allow ML engineers to focus on ML tasks, and provides tooling for easy access to data insights in order to accelerate ML experimentation. The technical challenges to achieve these goals can be categorized into the following areas, to be discussed next: 1) supporting the engineering teams, 2) supporting the machine learning life cycle, and 3) supporting the variety of ML frameworks and ML data.

2.1 Machine Learning Teams

The teams that build machine learning features comprise of ML engineers, data scientists, software engineers, and the product legal team. These distinct specialists need a data system to support them. ML engineers focus on finding the best ML model for the given problem, exploring different *features*[2] of the data, and experimenting with different combinations of data and ML models. Software engineers focus on building the toolset to support the end-to-end ML model training pipeline, starting from data cleaning, ETL, feature engineering, model training, model evaluation, and eventually model deployment. Data scientists work closely with both ML specialists and software engineers on feature engineering, annotation efforts, and data quality control. The product legal team oversees the legal terms of use of the data, privacy, compliance (such as GDPR[3]), and auditing. Existing ML data solutions do not provide an integrated data platform to support all of above mentioned collaboration. Instead, data silos are created by each discipline, and discrepancies arise due to the difficulties of maintaining consistency across data silos.

[2] Features are measurable properties or characteristics extracted or derived from raw data. Feature engineering is the process of obtaining features from raw data.

[3] GDPR stands for EU's General Data Protection Regulation.

Furthermore, acquiring and augmenting data is expensive and time consuming. Hence, it is imperative that teams are able to share their data within an organization under appropriate compliance and privacy constraints. In order to share the data, stakeholder identification on the source of the data is another key aspect for enterprises [8]. In conventional service models, data is encapsulated behind service interfaces and any change in data is not known to the consumers of the service. In machine learning, data itself is an interface which needs to be tracked and versioned. Hence, the ability to identify the ownership, the lineage, and the provenance of data is critical for such a system. Since data evolves through the life of the project, teams require data life cycle management features to understand how the data has changed. Again, silos make data sharing and discovery, data lineage and provenance tracking, version control, and access control difficult, if not impossible.

2.2 Machine Learning Life Cycle

The machine learning life cycle is highly iterative and experimental. While data is being injected during the data collection stage, data curation process starts incrementally with ETL pipelines to homogenize syntactic and semantic discrepancies, and the cleansed data is pipelined to feature engineering and annotation efforts. After the initial batches of features and annotations are ready, ML engineers start exploring different features and small-scale training experiments. Depending on the experiment results, existing feature engineering and annotation efforts may be paused, continued, or new data engineering effort may be started. Only after many, sometimes hundreds of, experiments does a promising mix of data, ML features, and a trained ML model emerge.

In order to assist tasks among these intertwined stages, a ML data platform needs to support

(1) a conceptual data model to describe both slowly changing data assets and volatile features,
(2) simple mechanisms for continuous data injection,
(3) a domain specific language for effective data and feature engineering,
(4) a hybrid data store, that are suited for large data volumes of stable raw assets and high concurrent updates on volatile data, with physical data layout designs that support data versioning, and are optimized for incremental updates, delta tracking among versions, and ML training access patterns,
(5) a data access interface integrated with major ML frameworks, allowing ubiquitous data access both within data centers as well as on the edge,
(6) explicit version management to ensure reproducibility of ML experiments,
(7) tracking for data lineage and provenance, and
(8) a toolset and a programming interface for data exploration and discovery.

It is common to conduct ML experiments across different mixes and matches of data and features. In any highly experimental process, it is essential that one can reproduce the results as needed. Existing ML solutions, which rely on users to bring their own data and data storage system, lack the support to easily reproduce the training results. The burden falls onto the users to manually track the dependencies among data versions, training tasks, and ML models. An ML data platform should be capable of tracking the dependencies among data, model, and code, as well as data lineage between raw data assets and the derived annotations and features. For example, in case of errors found in the source dataset, we can identify all the dependent and derived data, and notify their owners to regenerate the labels or annotations, or re-train the ML models.

In addition, ML experiments often begin with interactive data exploration before launching large scale training in the data center. An ML data platform should support both server-side data processing, such as predicate push-down and expression evaluation, as well as client-side remote streaming data access. The transition from training on laptop to training in data center should require no code changes in the applications, so the cycle from exploration to analysis to experimentation is as short as possible. Furthermore, similar to common large-scale data systems, automated sharding is essential to support distributed training by leveraging data parallelism, so is data caching in order to shorten the data distance.

2.3 Machine Learning Frameworks and Data

Many machine learning frameworks are available today, with new ones emerging at times. Most of them are free and open-source, and many of them are still under active development. Machine learning frameworks provide various learning algorithms and models for different problem domains in ML. Since each framework has strengths for different models, it is very common for an organization to utilize several frameworks within a given project (including different versions of the same framework). Most of the frameworks rely on the file system to access training data, with some frameworks offering additional data reader interfaces to make I/O more efficient, such as, TensorFlow and MXNet.

An integrated ML data platform should provide file I/O data access interfaces for common ML frameworks, with the extensibility to plug in custom record-level iterators or data readers for specific frameworks, such as `RecordIO` for MXNet and `TFRecordReader` for TensorFlow. These *data connectors* –

Solution	Integrated Data System	Model Training	Experimentation	Evaluation	Deployment
SageMaker	No (Bring your own data store: *e.g.*, S3, EBS, EFS)	Yes	Yes	Yes	Yes
Azure ML	No (Bring your own data store)	Yes	Yes	Yes	Yes
Cloud AI	No (Bring your own data store: *e.g.*, Google Cloud Storage)	Yes	Yes	Yes	Yes
TensorFlow	N/A	Yes	No	No	Yes
Keras	N/A	Yes	No	No	No
PyTorch	N/A	Yes	No	No	No
Michelangelo	No (HDFS data lake, Cassandra)	Yes	Yes	Yes	Yes
FBLearner Flow	N/A	No	Yes	No	Yes
Colaboratory	N/A	No	Yes	Yes	No
Databricks	N/A	No	Yes	Yes	No
MLdp	Yes	Integrated with major ML frameworks	Integrated with in-house execution clusters, as well as public elastic compute cloud	Integrated with an in-house solution	Integrated with an in-house solution

Table 1: Machine learning platform eco-System

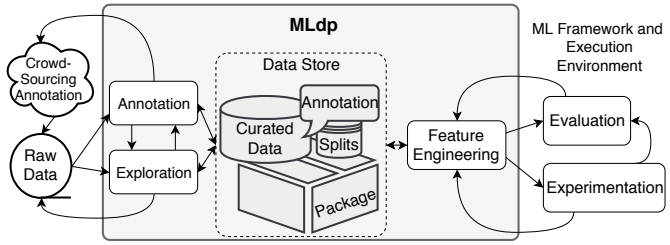


Figure 2: Integrated ML and data workflow: MLdp acts as data interface with data model, data access and life cycle management.

a framework specific implementation of the data access interface – should provide bi-directional integration. For example, data stored as proprietary format can be accessed directly from the ML data platform, while data stored in platform native format can be retrieved in a variety of formats, such as `TFRecords`.

In addition to variations in the data format, ML applications use wide varieties of rich data types, such as image, video, audio, document, text, numerical data, sensor data, *etc.* These rich data types should be first class citizens in the ML data platform so that the system can provide index-ability into these data in order to support efficient search and data exploration.

ML data can evolve in three dimensions – variety, volume and velocity. Typically, raw data assets are large in volume and change slowly. For example, individual images in a computer-vision ML (CV/ML) project will rarely change after they are acquired and stored, unless there is a problem in the image. On the other hand, features or annotations for a given image may change at a higher velocity but with lower volume. Different models may require different annotations. For example, bounding boxes on the objects in the images are required to train an object detection model, while textual labels of the scene categories are needed to train a scene recognition model. Additionally, if previous ML experiments did not yield satisfying results, it may trigger existing annotations to be refined, or to be redone in an entirely different way. A new ML project might also trigger new annotation efforts. All of the above lead to increased volume, variety and velocity of ML data. The ML data system needs to support agility in all three dimensions.

To sum up, based on the above observations and the experiences from using existing solutions, we believe the effort to design and implement a purpose-built data management system for ML is well deserved. As shown in Figure 2, MLdp is more than a data storage. The goals of MLdp are to assist ML tasks, such as annotation, exploration, data and feature engineering. MLdp acts as a data interface providing abstractions including a data model, data access interface, and data life cycle management for ML. MLdp also integrates with major ML frameworks' data access layer allowing ML pipelines to focus on experimentation and evaluation, leaving the onus of data access and management to MLdp.

2.4 Related Work

Thus far, we have mentioned many previous works which focus on different phases of the ML life cycle, such as ML algorithms, model training, and evaluation. ModelHub [17], ProvDB [16], and Model Governance [18] described systems that track the lineage of machine learning models. Compared to these works, MLdp aims to provide a dataset centric system to manage the life cycle of ML datasets, with the capability to track dependencies among data, models and training tasks.

DataLab [23] and SciDB [19] showcased systems that are designed for scientific experiments to track the provenance and versions of underlying data, which is similar to MLdp's data provenance and lineage tracking feature. In addition, MLdp provides data access interfaces integrated with popular machine learning frameworks and data compliance features tailored to assist ML data management. Petastorm [7] offers data access API and seamless integration with frameworks, such as TensorFlow. We share a similar vision and offer additional features such as versioning, compliance, and lineage tracking to ease the data management burden on the user.

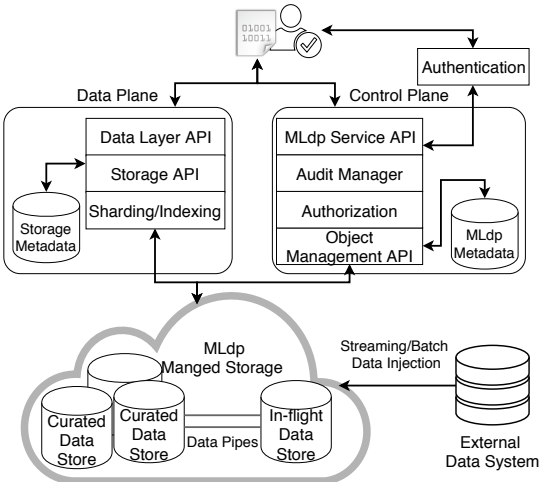


Figure 3: MLdp system architecture showing control plane and data plane and hybrid storage model.

Decibel [14] and ORPHEUSDB [22] are database systems that support data branching and versioning for structured relational data. Kayak [13], on the other hand, provides a data preparation framework with inferred catalog for loosely structured data in data lakes. MLdp takes the top-down approach by enforcing a minimalist data model to allow a centralized catalog for ease of discovery and sharing, and at the same time allows flexible schema for semi-structured data.

Systems such as DataHub [2] and Goods [8] inspired our work in terms of data sharing and discovery. MLdp offers additional compliance and privacy control features, with data version management and data access interfaces to fully integrate with ML life cycle.

Michelangelo is Uber's ML platform and shares similar mission statements as MLdp. From [21], however, the data lake approach, backed by HDFS, lacks data versioning and other compliance features.

Finally, Boehm et al [3, 4], and Kraska et al [11] proposed declarative domain specific languages for machine learning systems, named MLBase and SystemML, respectively. Currently, the design of the MLdp DSL focuses on data and feature engineering tasks, and we rely on low-level data API to integrate with ML frameworks. The reason is simply that most of the popular ML frameworks offer Python SDKs with easy to use control-flow configuration. Nevertheless, we are actively evaluating the practicality of declarative ML with MLdp's DSL.

3 SYSTEM ARCHITECTURE AND DESIGN

MLdp provides REST APIs and client SDKs for client-side data access, and a domain specific language (DSL) for server-side data processing. The service consists of *control plane* and *data plane* APIs to assist data management and data consumption. Figure 3 shows major components of MLdp at a high-level. Due to the length limit of the paper, we will not cover all the components. Instead, the discussions will focus on the designs that cover the following key requirements mentioned in Section 2:

- a conceptual data model to naturally describe raw data assets versus features/annotations derived from the raw data,
- a version control scheme to ensure reproducibility of ML experiments on immutable snapshot of datasets,
- data access interfaces that can be seamlessly integrated with ML frameworks as well as other data processing systems,
- a hybrid data store design that is well-suited for both continuous data injection with high concurrent updates and slowly-changing curated data,
- a storage layout design that enables delta tracking between different versions, data parallelism for distributed training, indexing for efficient search and data exploration, and streaming I/O to support both training on devices or in the data center, and
- a distributed cache to accelerate ML training tasks.

3.1 Data Model

3.1.1 Conceptual data model. MLdp provides four high-level concepts (objects) – *dataset*, *annotation*, *split*, and *package*. A dataset is a collection of entities that are the main subjects of ML training, *e.g.*, raw images for a CV/ML project. An annotation is a collection of labels and/or features describing the entities in its associated dataset, *e.g.*, bounding-boxes, object boundaries, or textual labels on raw images. A split consists of horizontal partitions of the foreign keys referencing its associated dataset. Commonly, a dataset may be split into a *training set*, a *testing set*, and a *validation set*. Both annotations and splits are *weak* objects. They do not exist by themselves. Instead, they are always associated with one dataset. A dataset can have multiple annotations and splits.

Figure 4 shows a sample dataset consisting of a collection of image files, an annotation that labels every image, and a split that divides the images into a training set and a test set.

The benefits of separating (or, *normalizing*) annotations and splits out from their datasets are multifold. It allows different ML projects to label or to split the data differently. For example, to train an object recognition model one may want to label the bounding boxes in the images, however, to train a scene classification model one may want to label the borders of each objects in the images. The normalization also allows the same ML application to experiment using different features or labels. Alternatively, one may want to split the dataset in different ways to experiment with different learning strategies.

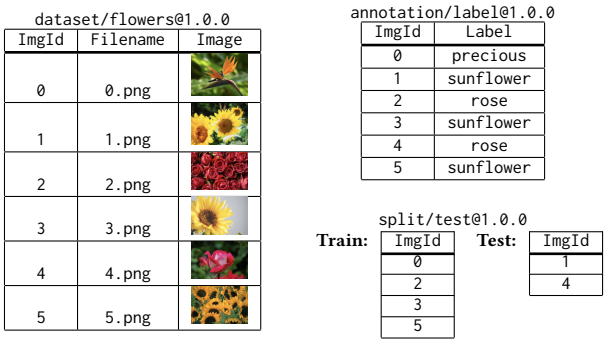


Figure 4: CV/ML data modeled as a dataset and its associated annotation and split.

Splits are similar to *partial indexes* in databases, while annotations are extracted features, or supplementary properties of the associated datasets. Both annotations and splits can evolve without changing the base dataset. In practice, dataset acquisition and curation are usually costly, labor intensive, and time consuming. Once a dataset is curated, it serves as the ground truth and will often be shared among different projects/teams. It is desirable that the ground truth does not change under their feet, but every project/team would like to label and organize the data based on their own needs and cadence.

Another reason for the normalization comes from legal or compliance requirements. Sometimes, labeling or feature engineering may involve additional data collection which is done under different contractual agreements than the base dataset. MLdp allows independent permissions and "Terms of Use" settings for datasets and annotations.

Packages are virtual objects, and provide a conceptual view over datasets, annotations, and/or splits. Like *views* in databases, packages offer a higher-level abstraction to hide the physical definitions of individual objects. For example, one can define an `IndoorImage` package over the union of the `ImageNet` [5] and the `OpenImages` [12] datasets by selecting all the indoor images.

3.1.2 Versioning and data life cycle management. MLdp offers a strong versioning scheme on all four high-level objects. Version evolutions are categorized into *schema*, *revision*, and *patch*, resulting in a three-part version number `<schema>.<revision>.<patch>` similar to the versioning scheme of `<major>.<minor>.<patch>` used in software.

A new `schema` version signals that the schema of the data has changed, so code changes may be required to consume the new version of the data. Both `revision` and `patch` version changes denote that the data is updated, deleted, and/or new entities have been added without schema changes. Existing applications should continue to work on new revisions or patches. It is worth noting that non-breaking changes to data do not necessarily imply application consistency. For example, adding new data in patch version could alter the results of an existing model training. A typical scenario of a patch is when a tiny fraction of the data is mal-formed during injection, re-touching those data results in a new patched version. It is essential that applications pin to a specific version to guarantee reproducibility.

The versioning scheme in MLdp offers applications the means to define their own version evolution policies. In contrast to common multi-versioned data systems where the versioning is implicit and system driven, the versioning in MLdp is explicit and data owner driven. MLdp's explicit version management allows different ML projects: 1) to share and to evolve the versions on their own cadence and needs without disrupting other projects, 2) to pin a specific version in order to reproduce the training results, and 3) to track version dependencies between data and trained models.

In machine learning, *data is an interface*. Any changes (either insertion, deletion or updates) in data should be versioned, just like software is versioned due to code changes.

To assist the life cycle management, each version of a MLdp objects can be in one of four states – *draft*, *published*, *archived*, and *purged*. The "draft" state offers applications the opportunity to validate the soundness of the data before transitioning it into the "published" state. Once published, the particular version of the data becomes immutable in order to guarantee the reproducibility of trainings. The only means to update published data is to create a new version of it. Once the data is expired or no longer needed, it can be transitioned into the "archived" state, or into the "purged" state to be completely removed from the persisted storage. For example, when a user opts out the user study, all the data collected on that user will be deleted resulting in a new patched version, while all the previous versions containing that user data will be purged.

3.1.3 Physical data model. MLdp represents its data in a tabular form without rigid schema requirements on all entities of a given object class. The only schema requirement is the primary key of a dataset, which uniquely identifies an entity in a dataset. In addition, it defines the foreign key in both annotations and splits to reference the associated entities in the datasets.

Columns in a table can be of scalar types, as well as *collection* types. Scalar types include number, string, date-time, rich multimedia types, and byte stream, while collection types include vector, set, and dictionary (document). MLdp tables are stored in column-wise fashion. Not only does the columnar layout yield a high compression rate, which in turn reduces the I/O bandwidth requirements, but it also allows adding and removing columns efficiently. In addition, MLdp tables are scalable distributed data structures, without the

restriction of main memory size. MLdp tables are interoperable with Apache Resilient Distributed Datasets (RDD), Apache DataFrames, Pandas, and R Data Frames.

ML datasets often contain a list of raw files. For example, to build a human posture and movement classification model, one entity in the dataset may consist of a set of video files of the same subject/movement from different angles, plus a `JSON` file containing the accelerometer signals. MLdp treats files as byte stream columns in the table. It is up to the system to decide whether to store the file contents in-row or off-row. Since most ML frameworks support file I/Os, MLdp provides streaming file accesses to those files, regardless of in-row or off-row storage layout. Moreover, MLdp allows user-defined access paths, such as primary indexes, secondary indexes, *partial indexes* (or, *filtered index*), *etc.*, for efficient data exploration and filtering.

3.2 Data Interface

MLdp offers a high-level data language and a low-level data API. The high-level domain specific language (DSL) offers a declarative programming paradigm that is aimed at assisting server-side batch data and feature engineering tasks in distributed and parallel execution environments closer to the data. The low-level data API offers an imperative programming paradigm that is well integrated with major ML frameworks and provides streaming I/O for client-side data processing. In order to shorten the distance to data for client-side data access, MLdp provides data center caching for teams that train models in custom execution clusters. MLdp cache service is a distributed cache service that can also be deployed to the same compute cluster where ML training occurs.

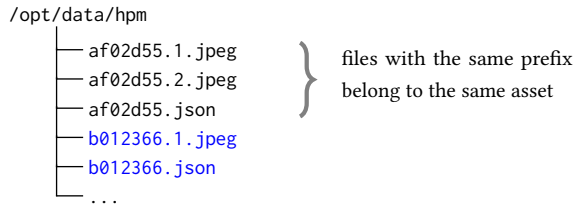
One of the key observations that applied ML teams realized, but is not surprising to the database community, is that significant amount of time was spent on data and feature engineering compared to the amount of time spent on training. While the DSL is still an experimental feature, we are confident that the DSL will help improve the productivity of ML teams. For the sake of clarity and the familiarity of the SQL language to the community, we will use the SQL-like variant of the DSL to illustrate the use cases. Another variant of the DSL which is more *Pythonic* by using the *builder pattern* is omitted due to the space limitation.

3.2.1 DSL. The declarative nature of the DSL allows users to describe the intent, and the programs will be compiled and optimized into execution graphs, which can be either executed locally or submitted to an elastic compute cluster for execution. The local execution mode provides testing and debugging before execution in the cluster. The query optimization techniques developed over past decades for relational databases, such as exploiting *interesting properties*, *view matching* and *index selection*, are applicable to MLdp's use cases.

Currently, we have integrated the DSL with Python, a popular language in the ML community, in order to support user code and ML frameworks written in Python. Instead of listing a full specification of the MLdp DSL, we will demonstrate the DSL in action by the following two examples. More examples can be found in Appendix A.

(1) Create a dataset from a set of raw images and `JSON` files describing the images.
(2) Use a user supplied ML model to create labels and publish them as a new version of an existing annotation.

Create a dataset The sample code[4] in Listing 1 shows how to create a dataset, named `human_posture_movement`, from the raw files under the path `"/opt/data/hpm"`. The files are organized as follows. The prefix to each file is the unique identifier to a logical entity (asset) in the dataset. An asset in this example contains a set of `JPEG` files, and the accelerometer readings in one `JSON` file.



The `CREATE dataset...WITH PRIMARY_KEY` clause defines the metadata of the dataset, while the `SELECT` clause describes the input data. The syntax `<qualifier>/<name>@<version>` denotes the *uniform resource identifier* (URI) for MLdp objects. In this example, the URI is `dataset/human_posture_movement` without the version, since `CREATE` statement always creates a new object with version starting from `1.0.0`.[5] The `FROM` sub-clause declares the variable binding, `_FILE_`, to each file in the given directory. The files are grouped by the path prefix, `_FILE_.NAME.split('.')[0]`, which is declared as the primary key of the dataset. Within each group of files, all the `JPEG` files are put into the `Images` collection column, and the `JSON` file is put into the `Accelerometer` column. The function, `DSL().Run()`, will compile and execute the statement.

```
# the following statement will create a
# DatasetTable 'hpm' with the
# columns (SessionId, Images, Accelerometer)
status = ml.data.DSL(
   "CREATE dataset/human_posture_movement
    WITH PRIMARY_KEY(SessionId) AS
```

[4] For the sake of simplicity, we omit settings such as permission, expiration, *etc.*, and focus on the data definition and data consumption grammar.
[5] The available `qualifiers` are: `dataset`, `annotation`, `split`, and `package`, For weak objects, *i.e.*, annotations and splits, full qualifiers in the form of `dataset/<name>@<version>/(annotation|split)` are required.

```
    SELECT SessionId,
        CASE WHEN _FILE_.EXT='jpeg'
            THEN COLLECTION(_FILE_) AS Images
        CASE WHEN _FILE_.EXT='json'
            THEN _FILE_ AS Accelerometer
        END
    FROM FILES IN './data/hpm' AS _FILE_
    GROUP BY _FILE_.NAME.split('.')[0] AS SessionId").Run()
```

Listing 1: Creating a dataset from files under a local folder

Use a pre-trained model to create new labels The sample code in Listing 2 shows how to create a new version of an existing annotation on the `human_posture_movement` dataset. The reserved symbol, `@`, is used to specify a particular version of an object. `ALTER...WITH REVISION`[6] will create a *revision* based on the specified version. In this example, the new version will be `human_activity@1.3.0`. The `ON` sub-clause specifies the version of the dataset which this annotation references to. The `SELECT...FROM` clause defines the input data source. `SessionId` is the foreign key to the parent dataset. This example also demonstrates user code (Turi Create [1]) integration with the MLdp DSL. User code dependencies must be declared by the `import` statements.

```
# the following statement will create a new version
# of the annotation, 'human_activity', with
# the columns (SessionId, Activity)
status = ml.data.DSL(
   "import turicreate as tc;
    ALTER annotation/human_activity@1.2.0
        WITH REVISION, FOREIGN_KEY(SessionId)
        ON dataset/human_posture_movement@1.0.0 AS
    SELECT SessionId,
        tc.load_model('dfs://ml/activity_classifier.ml').
            predict(Accelerometer) AS Activity
    FROM human_posture_movement@1.0.0").Run()
```

Listing 2: Creating a new version of an annotation that contains automatically generated labels by a pre-trained ML model

The pre-trained model, `activity_classifier.ml`, is used to predict the activity based on the accelerometer signals, and the prediction is stored as a label in the `Activity` column.

3.2.2 Low-level data primitives. MLdp's data access primitives provide direct access to data via both streaming File I/Os and table API. The streaming on-demand enables effective data parallelism in distributed training.

Streaming File I/O It is common that ML datasets contain collections of raw multimedia files that the ML models directly work on. The MLdp client SDK allows applications to *mount* MLdp objects, and the mount point exposes those raw files in a logical file system. The `mount` command implements data streaming on-demand. That is, physical blocks containing the files or the portion of a table being accessed are transmitted to the client machine in time. Currently, rudimentary prefetching and local caching are implemented in the `mount`-client. Since most, if not all, of the ML frameworks support File I/Os in their data access abstraction, the mounted logical file system provides a basic integration with most of the ML frameworks from day one. To support ML applications running on the edge, MLdp also provides direct file access via REST API.

Listing 3 shows a Python application that mounts the OpenImages dataset [12], and performs *corner detection* on each image by directly reading the image files.

```
# mount the dataset
data = ml.data.mount('dataset/OpenImagesV4@1.0.0', './mnt')

# data.raw_file_path points to the mount-point folder
# that contains the raw files;
for entry in scandir.scandir(data.raw_file_path):
    # Harris Corner Detector
    img = cv2.imread(entry.path, cv2.IMREAD_GRAYSCALE)
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    gray = np.float32(gray)
    dst = cv2.cornerHarris(gray,2,3,0.04)
```

Listing 3: Mounting the OpenImages dataset

Table API As described in Section 3.1.3, MLdp stores data as tables in the columnar format, with the support of user-defined access paths (*i.e.*, the secondary indexes). The table API allows applications to directly address both user tables and secondary indexes.

Listing 4 shows a simple application which uses a secondary index to locate data of interest, and then performs a key/foreign-key join to retrieve the images from the primary dataset table for *image thresholding* processing.

```
# mount the dataset
data = ml.data.mount('dataset/OpenImagesV4@1.0.0', './mnt')

# use the secondary index to select images of interest
img_class_idx = data.indexes['img_class']
person_class = img_class_idx.where('Category'='Person')

# fetch the data by joining back to the primary index
person_data = data.primary_table.join(person_class,
    on='ImageId')

# now load all the person images for thresholding
for row in person_data:
    img = cv2.imread(row['filename'], IMREAD_GRAYSCALE)
    thresh1 = cv2.threshold(img,127,255,THRESH_BINARY)
    thresh2 = cv2.threshold(img,127,255,THRESH_BINARY_INV)
    thresh3 = cv2.threshold(img,127,255,THRESH_TRUNC)
```

Listing 4: Using Table API with secondary indexes to access data

[6] Similar syntax is available for dataset, split and package version updates. Available options for version updates are `SCHEMA`, `REVISION`, and `PATCH`. Moreover, for the sake of brevity, the examples will omit the full-qualifier for annotations and splits, when the full-qualifier is obvious in the context.

Distributed training A common distributed training configuration involves multiple worker nodes which train the same model on different partitions of the data. Those workers will update their parameters on one or more parameter servers, which will then broadcast the aggregated parameters to the workers for subsequent training iterations.

Many ML frameworks support distributed training natively by allowing a user specified configuration describing how workers and parameter server(s) are allocated, and how data will be partitioned and shuffled to each worker. In other cases without framework support, one has to implement the sharding, shuffling, and coordination by oneself. In the following, we will illustrate how to integrate MLdp with distributed training using TensorFlow [20]. Another example that uses MXNet [15] will be listed in Appendix B.

Listing 5 shows how each worker in TensorFlow accesses a slice of input data by its `task_index`. Since MLdp streams the data on-demand, each worker will only incur the I/O costs proportional to the actual data loaded.

```
# mount the dataset
data = ml.data.mount('dataset/OpenImagesV4@1.0.0', './mnt')
filenames = data.primary_table.select_columns(['id','path'])

input_partition = tf.strided_slice(filenames,
    [task_index],
    [filenames.num_rows()],
    strides = [num_workers])

# training logic here ...
```

Listing 5: Distributed training integrated with TensorFlow

3.3 Storage and Data Layout Design

MLdp's design benefits from many key learnings found in database systems. In this section, we will discuss MLdp's storage layer design in order to meet the following requirements for supporting ML life cycle.

- A hybrid data store that supports both high velocity updates at the data curation stage and high throughput reads at the training stage.
- A scalable physical data layout that can support ever-growing data volume, and efficiently record and track deltas between different versions of the same object.
- Partitioned indices that support dynamic range queries, point queries, and efficient streaming on-demand for distributed training.

3.3.1 Hybrid Data Store. At early stages of data collection and data curation, raw data assets and features are stored in an in-flight data store, as shown in Figure 3. The in-flight data store uses a distributed key-value store that supports efficient in-situ updates and appends concurrently at a high velocity. The in-flight store only keeps the current version of its data. Snapshots can be taken and published to MLdp's curated data store, which is a versioned data store based on a distributed cloud storage system. The curated data store is read-optimized, and yet supports efficient append-only updates and sub-optimal in-situ updates based on *copy-on-write*. Changes to a snapshot in the curated store will result in a new version of the snapshot. A published snapshot will be kept in the system to ensure reproducibility of ML experiments until it is archived or purged. How MLdp optimizes both workloads in different data stores lies in the physical data layout design, which will be discussed in detail in Section 3.3.2.

Data movement between the in-flight and curated data stores is managed by a subsystem, named "data-pipe". Each *logical* data block in both data stores maintains a unique identifier, a *logical* checksum, and a timestamp of last modification. Data-pipe uses this information to track deltas between different versions of the same dataset.

Matured datasets can be removed from the in-flight store after storing the latest snapshot in the curated store. On the other hand, if needed, a copy of a snapshot can be moved back to the in-flight store for further modification at a high velocity and volume. After the modification is complete, it can be published to the curated data store as a new version.

Despite the multiple data stores, MLdp offers a unified data access interface. The visibility of the two different data stores is purely for administrative reasons to ease the management of data life cycle by the data owners. It is also worth noting that using data from the in-flight store for ML experiments is discouraged, since the experiment results may not be reproducible due to the fact that data in the in-flight store may be overwritten.

3.3.2 Scalable Data Layout. MLdp stores its data in partitions, managed by the system. The partitioning scheme cannot be directly specified by the users. However, users may define a sort key on the data in MLdp . The sort key will be used as the prefix of the range partition key. Since there is no uniqueness requirement on the user-defined sort key, in order to provide a stable sorting order based on data injection time, the system will append timestamp to the partition key. If no sort key is defined, the system will automatically use the hash of the primary key as the range partition key. The choices of the sort keys depend on the sequential access patterns to the data, similar to the problem of *physical database design* in relational databases.

In case of data skew in the user-defined sort key, the appended timestamp column helps alleviate the partition skew problem. The timestamp provides sufficient entropy to split a partition either based on heat or based on volume. In addition, range partitioning will allow the data volume to scale

out efficiently without the issue of global data shuffling that naive hash partition schemes suffer from.

Each logical partition is further divided into a sequence of physical data blocks. The size of the data blocks is variable and can be adjusted based on access patterns. Both splits and merges of data blocks are localized to the neighboring blocks, with minimum data copying and movement. This design choice is particularly influenced by the fact that published versions of the MLdp data are immutable. Version evolutions typically touch a fraction of the original data. With the characteristics of minimum and localized changes, old and new versions can share common data blocks whose data remain unchanged between versions.

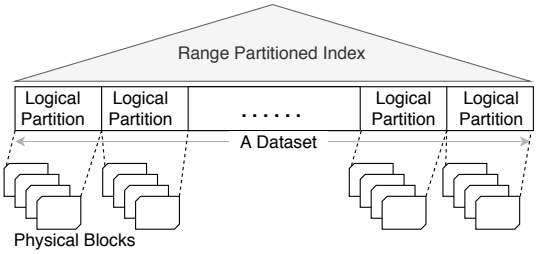


Figure 5: MLdp physical data layout.

Figure 5 illustrates the data layout based on range partitioning. MLdp's storage engine maintains an additional index on the range partition key to efficiently locate a particular partition/data block based on user predicates. When a new version is created with incremental changes to the original version, only the affected data blocks are created with *copy-on-write*. As shown in Figure 6, updates trigger a copy of the original data block, followed by a split, and a new version is created with minimum data movement.

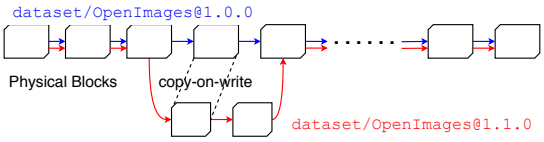


Figure 6: Create a new version of a dataset via copy-on-write.

3.3.3 Performance Optimization. MLdp's data layout design enables the following optimization strategies that benefit typical ML access patterns.

Data Parallelism Typical data and feature engineering tasks are highly parallelizable. The system can exploit the interesting partition properties as well as the existing partition boundaries to optimize the task execution. In addition, for distributed training where data is divided into subsets for individual workers, the partitioned input provides a good starting point before the data needs to be randomized and shuffled.

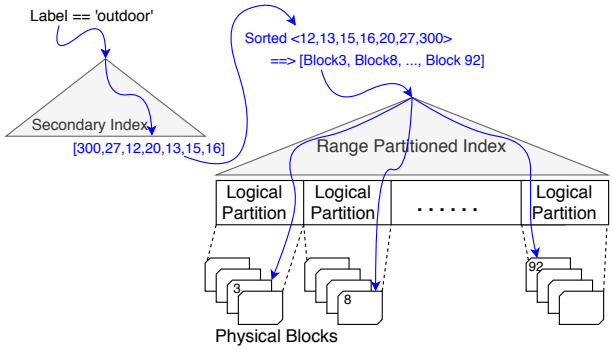


Figure 7: Using secondary index to map keys into data block IDs and retrieve data of interest.

Streaming On-Demand ML training experiments may target only a subset of the entire dataset, *e.g.*, to train a model to classify the dog breeds, we might be only interested in the dog images from the entire computer vision dataset. After identifying the image IDs, the actual images might be scattered across many partitions, the data block layout design will allow MLdp's client to stream only those data blocks of interest. In addition, many training tasks have a predetermined access sequence, a fine-tuned data block size gives the system a fine-grained control on prefetching optimization. In addition, streaming I/O improves the resource utilization, *esp.*, the highly contended GPUs, by reducing the idle-time waiting for the entire training data. Before the streaming I/O feature was available, each training task had a long initial idle-time, busy-waiting for the entire data to be downloaded.

Range Scan and Point Query Each data block and partition contains aggregated information about the key ranges within. The data blocks are linearly linked to support efficient scans, while the index over the key ranges allows efficient point queries.

Secondary Indexes MLdp allows users to materialize search results, similar to *materialized views* in databases. Secondary indexes are simpler variations of generic materialized views. The leaf-nodes of the secondary indexes store a collection of partition keys. Since MLdp employs range partitioning, the system can easily sort and map the keys into partition IDs and data block IDs without duplication. This further improves the I/O throghput and latency by batching multiple key requests into a single block I/O. Figure 7 illustrates the use of a secondary index to batch block I/Os.

3.4 Distributed Cache

Most ML applications perform client-side data processing, *i.e.*, bringing data to compute. In order to shorten the data distance, MLdp provides a *transparent* distributed cache in the data center *collocated* with the compute cluster of ML tasks. The cache service is transparent to applications, since applications will not directly address the cache service endpoint,

instead they always connect to the MLdp API endpoint. If MLdp finds a cache service that is collocated with the execution cluster where the application is running, it will notify the MLdp client to redirect all subsequent data API calls to the cache cluster. The MLdp client has a built-in fail-safe in case the cache service becomes unavailable, the data API calls fall back to the MLdp service endpoint.

Currently, many different execution environments are used by different teams, and more are being added as ML projects/teams proliferate in various domains. The cache service is designed to be deployable to any virtual cluster environment. The goal of such a design choice is to be able to setup the cache service as soon as the execution environment is ready.

Another design goal of the MLdp cache service is to achieve *read scale-out*, in addition to the reduction of data latency. The system throughput increases by scaling out existing cache services, or by setting up new cache deployments. We made a conscious design choice that the MLdp cache service only caches read-only snapshots of the data, *i.e.*, the published versions of data. The decision favors a simple design to guarantee strong consistency of the data. The anomalies caused by the *eventual consistency* model impede the reproducibility guarantee. If mutable data were also cached, in order to ensure transactional consistency of the cached data, data under higher volume of updates not only will not benefit from caching, but the frequent cache invalidation puts counterproductive overheads to the cache service.

4 FUTURE WORK

The inception of MLdp is driven by the lessons learned from using previously available ML solutions, which led to data silos. As the complexity and the limitations imposed by data silos start to become the bottleneck of ML projects, it also becomes evident that we need to build a data platform that integrates with the ML workflow, and provides a unified data abstraction through all steps of the life cycle. So far, we discussed many of the important design aspects of MLdp, the design goals, and the rationale behind them. However, the system is far from complete. In this section, we will call out future work, which is categorized into the following areas: *data discoverability*, *system improvements*, and *eco-system integration*. For each of the three areas, we will highlight the strategic features with the goal of maximizing the productivity of ML teams.

4.1 Data Discovery and Exploration

ML projects often have a time-consuming human-in-the-loop phase of finding the right data for training tasks. As more datasets have been loaded into MLdp, new projects may benefit from exploring the existing data, instead of starting a new data acquisition journey. MLdp provides basic catalog search capabilities, such as queries by names, descriptions, samples of the datasets, *etc.* Data discoverability for ML goes beyond the basic catalog search.

Discoverability is a key requirement for ML data platforms. Catalog searches allow users to locate "*known*" information, and data exploration allows users to uncover "*unknown*" information. For example, given is an ML project to train a model that can predict dog breeds. Catalog searches can quickly locate datasets that contains images, or even animal images if a textual label of "*animal family*" is available in the dataset. The next step may be to find out all the images that might seem to have a dog-like shape in it, or to understand the distribution of certain properties of the images, such as lighting, the camera angles of the shots, colors of the coats of the animals in the images, *etc.* Some of these data exploration activities are ML tasks by themselves with the help of pre-trained models. Oftentimes, it also involves labor-intensive crowd-sourced annotation efforts to find the bounding boxes, object boundaries, *etc.*, before we can find out the data distribution of the image features which may in turn prompt refinements in the annotation efforts. Even in the midst of training experiments, ML specialists may need to understand if the training dataset has a data skew on certain breeds, or is missing others, leading to a learning skew.

Although much of the work involved to support data discovery is domain specific, there are many system-level features needed in order to provide an ease-of-use programming experience and fast turn-around time for interactive data exploration. One example is to have tooling support, such as interactive data studio, data visualization, *etc.* Another example is to start with a common ML domain, *e.g.*, computer-vision ML (CV/ML), by building a background process to learn the ontology of the dataset. By materializing the pre-computed distribution of the data assets using the ontology, the system can provide real-time data exploration on common queries.

After the data of interest is identified, the user may want to publish it as a materialized dataset into MLdp. Since the data may come from different datasets with different formats, it becomes evident that a unified presentation of the heterogeneous data format is required. The DSL should support user-defined functions, user-defined data type plug-ins to help with data transformation.

4.2 System Improvements

Two areas of system improvement to boost the productivity of ML teams are: 1) to reduce data latency and to increase data throughput, and 2) to reduce human-in-the-loop time.

As discussed in Section 3.4, MLdp's cache service offers a basic *object-based* read-cache to reduce data distance and

to alleviate cross data center network bandwidth bottleneck with collocated cache services. Object-based caching will cache data at the object level, *i.e.*, the cached data map directly to a persisted object in the data system. For example, if the dataset `OpenImages@1.0.0` is cached, any access to the dataset results in a cache hit. We plan to enhance the MLdp cache service by allowing *intent-based* caching. That is, the system can dynamically decide to cache frequently used query results. Similar to view matching in databases, at data access time, MLdp's cache system will try to match the user query with the cached query definitions. If the result of the user query is subsumed by that of a cached query, then the user query will directly access the cached data (with optional residual predicates, if applicable).

Another improvement is to have better prefetching prediction and local buffering. Effective prefetching can further improve read latency by pipelining data I/Os with training computations. However, in model training, effective prefetching is challenging, since inputs are usually required to be randomly shuffled before feeding into the training model. Reasoning an effective prefetching policy and being adaptive at run-time is a work in-progress with our ML partner teams.

The second area of system improvement is to have full integration of the DSL with commonly used parallel data processing systems, such as Spark or PySpark, to accelerate the heavy-lifting data and feature engineering tasks. Our experiences show that data and feature engineering tasks contribute a significant portion of time to the ML project. We believe a declarative programming paradigm of the DSL will allow 1) users to focus on their business logic without dealing with the nuisances of large scale data processing, and 2) systemic optimizations for parallel data processing.

4.3 Eco-System Integration

As discussed in Section 3.2, many frameworks define their own data formats, such as `TFRecord` in TensorFlow, and `RecordIO` in MXNet, *etc.* It is an on-going effort to have MLdp integrated deeper with major ML frameworks so that user applications remain compatible at the higher API level. The goal is to have minimum code changes to the user applications so that data migration to MLdp is smooth.

There are two design considerations. The first consideration is to keep the MLdp data stored in MLdp's own format, and use a format connector to convert into target format, *e.g.*, `TFRecord`, at run-time. The second consideration is to store MLdp data in the native target format, *i.e.*, `TFRecord`. In this case, we only need a lightweight connector, which directly hydrates the on-disk data into in-memory `TFRecords`.

The potential benefit of the first approach is that MLdp's native format becomes the mediator format, and the number of connectors needed is a linear function to the number of ML frameworks that need to be supported. However, the drawback is that the conversion happens every time at run-time. On the other hand, the benefit of the second approach is that for ML applications that use TensorFlow, it incurs almost no run-time overhead. However, the drawback is that for ML applications that use other ML frameworks than TensorFlow, they have to convert the data at run-time. The number of connectors needed will be quadratic to the number of supported ML frameworks.

Another important on-going integration is the workflow integration between MLdp and annotation platforms. Large scale annotation efforts are usually crowd-sourced using external third-party storage as staging areas for data export and import. As mentioned earlier that annotation efforts are usually interleaved with data exploration iteratively. Having a seamless smooth workflow between annotation and data exploration will promote productivity of the ML teams.

5 CONCLUDING REMARKS

Machine learning reemerges and flourishes within the last decade. It combines all four scientific paradigms – *theoretical science*, *experimental science*, *computational science*, and *data-intensive science*. [10] Machine learning is the study of mathematical models that leverage computational advances to learn the patterns within mass data, and repeat that experiment until we reach satisfactory results. Data, at the core of the fourth paradigm, plays a critical role to the rate of convergence. From past experiences, we observe that existing ML solutions do not offer well-integrated data systems into the machine learning workflow. Simple storage systems or distributed file systems are used as passive data stores, resulting in data silos and leaving every step in the ML workflow exposed to physical data dependence. The rate of innovation is then hampered by the intertwined physical dependencies. Furthermore, the silos create a barrier to offer holistic solutions to data management.

In this paper, we propose a purpose-built ML data platform whose design centers around ML data life cycle management and ML workflow integration. Its data model enables collaboration through data sharing and versioning, innovation through independent data evolution, and flexibility through minimalist schema requirements. Its data interface design aims to provide ubiquitous data access, including ease of discovery, ease of use, and interoperability.

We believe a data system with these properties will enable faster ML innovation for organizations. There are still many challenges yet to be addressed including, but not limited to, declarative ML data language, system internal optimization, ML framework integration, *etc.* Finally, we hope that a system like MLdp can provide a basis for further exploration of how to support machine learning needs with data systems.

REFERENCES

[1] Apple. Turi create. https://github.com/apple/turicreate/, 2018; accessed November 28, 2018.

[2] A. P. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran. Datahub: Collaborative data science & dataset version management at scale. *CoRR*, abs/1409.0798, 2014.

[3] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, and S. Tatikonda. Systemml: Declarative machine learning on spark. In *PVLDB*, volume 9, 2016.

[4] M. Boehm, A. V. Evfimievski, N. Pansare, and B. Reinwald. Declarative machine learning - a classification of basic properties and types. In *CoRR, abs/1605.05826*, 2016.

[5] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings on Computer Vision and Pattern Recognition*. IEEE Computer Society, June 2009.

[6] Facebook. Introducing FBLearner Flow: Facebook's AI backbone. https://code.fb.com/core-data/introducing-fblearner-flow-facebook-s-ai-backbone/, 2018; accessed November 28, 2018.

[7] R. Gruener, O. Cheng, and Y. Litvin. Introducing Petastorm: Uber ATG's data access library for deep learning. https://eng.uber.com/petastorm/, 2018; accessed November 28, 2018.

[8] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing Google's datasets. *SIGMOD*, 2016.

[9] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, Feb 2018.

[10] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, October 2009.

[11] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[12] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, T. Duerig, and V. Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv:1811.00982*, 2018.

[13] A. Maccioni and R. Torlone. Crossing the finish line faster when paddling the data lake with kayak. *Proceedings of the VLDB Endowment*, 10(12):1853–1856, 2017.

[14] M. Maddox, D. Goehring, A. J. Elmore, S. Madden, A. Parameswaran, and A. Deshpande. Decibel: The relational dataset branching system. *Proceedings of the VLDB Endowment*, 9(9):624–635, 2016.

[15] Apache MXNet. Mxnet data api. https://mxnet.incubator.apache.org/versions/master/api/python/io/io.html, 2018; accessed November 28, 2018.

[16] H. Miao, A. Chavan, and A. Deshpande. Provdb: Lifecycle management of collaborative analysis workflows. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics*, page 7. ACM, 2017.

[17] H. Miao, A. Li, L. S. Davis, and A. Deshpande. Towards unified data and lifecycle management for deep learning. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 571–582. IEEE, 2017.

[18] V. Sridhar, S. Subramanian, D. Arteaga, S. Sundararaman, D. Roselli, and N. Talagala. Model governance: Reducing the anarchy of production {ML}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 351–358, Boston, MA, 2018. USENIX Association.

[19] M. Stonebraker, P. Brown, A. Poliakov, and S. Raman. The architecture of scidb. In J. Bayard Cushing, J. French, and S. Bowers, editors, *Scientific and Statistical Database Management*, pages 1–16, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[20] TensorFlow. An open source machine learning framework for everyone. https://www.tensorflow.org/, 2018; accessed November 28, 2018.

[21] Uber. Meet Michelangelo: Uber's machine learning platform. https://eng.uber.com/michelangelo/, 2017; accessed November 28, 2018.

[22] L. Xu, S. Huang, S. Hui, A. J. Elmore, and A. Parameswaran. Orpheusdb: a lightweight approach to relational dataset versioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1655–1658. ACM, 2017.

[23] Y. Zhang, F. Xu, E. Frise, S. Wu, B. Yu, and W. Xu. Datalab: A version data management and analytics system. In *Proceedings of the 2Nd International Workshop on BIG Data Software Engineering*, BIGDSE '16, pages 12–18, New York, NY, USA, 2016. ACM.

Appendix A DSL EXAMPLES

Below are three further examples of using DSL for creating a split and a package, and training an activity classifier.

Create a split The sample code in Listing 6 creates a split of the `human_posture_movement` dataset into a training subset and a testing subset. The `ON` clause defines the dataset which this split references to, and the `FROM` clause specifies the data source, which is the join between `human_activity@1.3.0` and `human_posture_movement@1.0.0`. The optional `WHERE` clause specifies the filter conditions. The split, named `outdoor`, only contains entities labelled as one of the three outdoor activities. Note that a split does not contain any user defined columns. Instead, it only contains the reference key (foreign key) to the corresponding dataset. As a result, the `SELECT` clause is not supported in the `CREATE split` or `ALTER split` statements. Finally, the argument, `perc = 0.8`, to the function `RANDOM_SPLIT_BY_COLUMN()` specifies that 80% of entities will be included in the training set, and the rest will be included in the testing set.

```
# the following statement will create
# the split 'outdoor', which contains two partitions
# 'train' and 'test'
status = ml.data.DSL(
   "CREATE split/outdoor(train, test)
    WITH RANDOM_SPLIT_BY_COLUMN(column='SessionId',
         perc=0.8)
    ON dataset/human_posture_movement@1.0.0
    FROM human_posture_movement@1.0.0 JOIN
         human_activity@1.3.0 ON SessionId
    WHERE Activity in {'biking', 'jogging',
         'hiking'}").Run()
```

Listing 6: Creating a split

Create a package The code in Listing 7 creates the package, named `outdoor_activity`, which is defined as a virtual view over a three-way join among `human_posture_movement`, `human_activity`, and `outdoor` on the primary key and foreign keys. The `SELECT` list defines columns of the view. The package contains two tablets – `train_data` and `test_data`.

```
# the following statement will create
# a package 'outdoor_activity'
status = ml.data.DSL(
   "CREATE package/outdoor_activity(train, test) AS
    SELECT SessionId, Images, Accelerometer, Activity
    FROM (dataset/human_posture_movement@1.2.0 AS d
      JOIN annotation/human_activity@1.3.0 AS a
      ON d.SessionId = a.SessionId)
      JOIN split/outdoor@1.0.0 AS s
      ON d.SessionId = s.SessionId").Run()
```

Listing 7: Creating a package

Train an activity classifier Listing 8 shows a simple model training example. It loads the package, `outdoor_activity`, into both `train_data` and `test_data` tablets. Next, it creates and trains the model using the training data. Finally, it evaluates the model performance using the testing data.

```
# the following statement will load the package
# and train the activity_classifier using the
# training data

train_data, test_data = ml.data.DSL(
   "SELECT SessionId, Images, Accelerometer, Activity
    FROM package/outdoor_activity@1.0.0").Run()

model = turicreate.activity_classifier.create(
   train_data,
   session_id = 'SessionId',
   target = 'Activity')

metrics = model.evaluate(test_data)
```

Listing 8: Train an activity classifier

From the above examples, we can see that the DSL leverages most of the SQL expressiveness to simplify the tasks of data operations. One of the open design issue for MLdp DSL in the near future is to incorporate *declarative machine learning* into the language. [3, 4]

Appendix B AN EXAMPLE WITH MXNET

The following example uses MXNet [15] data loading API – `ImageRecordIter`. In MXNet, one can use `ImageRecordIter` to specify what partition of the input to read. After mounting the targeted MLdp dataset in command-line, the `im2rec.py` tool from MXNet is used to generate the list of input files. The file list is then used by `ImageRecordIter`, together with the parameters – `part_index` and `num_parts`, to pipeline the partition of the input data that each worker trains on, as shown in Listing 9.

```
# MXNet provides tools/im2rec.py to generate training
# and validation lists, 'openimages_train.lst' and
# 'openimages_val.lst', by the following command:
# >>> python tools/im2rec.py openimages ./data --list True
     --recursive True --train-ratio .85 --exts .png

store = kv.create('dist')
trainer = gluon.Trainer(..., kvstore = store)

# at each worker
data = ImageRecordIter(
    path_imglist = "./openimages_train.lst",
    num_parts = store.num_workers,
    part_index = store.rank,
    ...)
```

Listing 9: Distributed training integrated with MXNet