The Hopsworks Feature Store for Machine Learning

Javier de la Rúa Martínez[1,2], Fabio Buso[1], Antonios Kouzoupis[1], Alexandru A. Ormenisan[1], Salman Niazi[1], Davit Bzhalava[1], Kenneth Mak[1], Victor Jouffrey[1], Mikael Ronström[1], Raymond Cunningham[1], Ralfs Zangis[1], Dhananjay Mukhedkar[1], Ayushman Khazanchi[2], Vladimir Vlassov[2], Jim Dowling[1,2]
[1]Hopsworks AB, Stockholm, Sweden
[2]KTH Royal Institute of Technology, Stockholm, Sweden

ABSTRACT

Data management is the most challenging aspect of building Machine Learning (ML) systems. ML systems can read large volumes of historical data when training models, but inference workloads are more varied, depending on whether it is a batch or online ML system. The feature store for ML has recently emerged as a single data platform for managing ML data throughout the ML lifecycle, from feature engineering to model training to inference.

In this paper, we present the Hopsworks feature store for machine learning as a highly available platform for managing feature data with API support for columnar, row-oriented, and similarity search query workloads. We introduce and address challenges solved by the feature stores related to feature reuse, how to organize data transformations, and how to ensure correct and consistent data between feature engineering, model training, and model inference. We present the engineering challenges in building high-performance query services for a feature store and show how Hopsworks outperforms existing cloud feature stores for training and online inference query workloads.

1 INTRODUCTION

In 2017, Uber announced a platform called Michelangelo [51] that creates and manages data for training and inference for their machine learning (ML) models, massively reducing the time it takes for them to put and maintain ML models in production. Michelangelo introduced their feature store for machine learning, Palette, as a new class of data platform that provides high-performance read and write of feature data for different workloads - from feature engineering to model training to model inference. Palette is a dual-database system, with historical feature data stored in a data warehouse and the latest feature data (used by online models) stored in a key-value store. All existing commercial and open-source feature stores follow this same dual-database architecture, as a single Hybrid transaction/analytical processing (HTAP) database has not yet been shown to be capable of running the high throughput, low latency workloads required by online models, such as personalized recommendations [28, 49], and the massive data volumes and high read bandwidth required when training models.

SIGMOD '24, June 09–15, 2024, Santiago, Chile
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Feature centralization (governance, security, search) and reuse is another core capability of feature stores. Facebook reported that in their feature store "most features are used by many models", and that the most popular 100 features are reused in over 100 different models [31]. The benefits of feature reuse include higher quality features through increased usage and scrutiny, reduced storage costs - Facebook reported using 1900 servers to store online feature data for just 27% of their use cases - and reduced feature development and operational costs, as models that reuse features do not need new feature pipelines.

In this paper, we introduce the Hopsworks Feature Store, a highly available data platform for managing feature data for ML that supports a mix of transactional, point-in-time analytical, and semantic search queries. Hopsworks Feature Store addresses the data challenges in building ML systems and its contributions include:

(1) support for collaborative development of ML systems based on centralized, governed access to feature data, along with a new unified architecture for ML systems as feature, training and inference pipelines;
(2) feature reuse through computing features once and reusing them across multiple models;
(3) support for multiple feature computation frameworks - batch, streaming, and request-time computation - enabling ML systems to be built based on their feature freshness requirements;
(4) a new taxonomy for data transformations based on the type of feature they compute (a) reusable (model-independent) features, (b) model-dependent features, and (c) request-time features. Our taxonomy requires additional framework support to prevent skew between data transformation implementations reused in two or more ML pipelines;
(5) a query model for how to create training data, without future data leakage, using an AsOf Left (outer) join, as well as an implementation of a query service based on Arrow Flight [10], DuckDB [39], and Apache Hudi [21] that outperforms publicly accessible managed feature stores in public clouds;
(6) a query model for how to read precomputed features for online inference as a set of parallel Left (outer) joins as well as pushdown support for Left joins in RonDB [1];
(7) a query model for how to find similar features using vector embeddings.

We discuss its novel design choices to address challenges 1-7. Challenges 5 and 6 imply support for mixed columnar and row-oriented workloads, with high throughput batch reads of feature data, and low latency reads for online inference. To tackle this, Hopsworks was built with a dual database architecture with HopsFS-S3 [23] and/or external data warehouses as the columnstore (called the offline store), and RonDB as its rowstore (called the online store). But Hopsworks now also has a vector database to enable another common ML workload - similarity search. Hopsworks was the first open-source feature store, and currently has thousands of users, with customers that store over a petabyte of data on a single Hopsworks cluster.

2 ML SYSTEMS WITH A FEATURE STORE

ML systems are platforms that manage data and models, help transform data into features, and use ML models and features to make predictions. Feature stores support three different types of ML system:

- *interactive ML systems* make predictions in response to user requests. They can combine features computed from request parameters with precomputed features from the feature store (providing history and context to user requests). They can make sure features are fresh (less than a few seconds old) by computing features on-demand from request input data or by updating precomputed features in the feature store using stream processing;
- *batch ML systems* run on a schedule, and read a batch of precomputed features from the feature store, download a trained model from a model registry, and make predictions for all rows in the batch using the model. The predictions are typically stored in some downstream database (inference store), to be later consumed by ML-enabled applications;
- *stream processing ML systems* use an embedded model to make predictions on streaming feature data, computed on-demand. They may also enrich their feature data with historical or contextual precomputed features retrieved from a feature store.

2.1 Feature, Training, and Inference Pipeline Architecture

Hopsworks was the first API-based feature store and it enabled a new ML system architecture based on independent ML pipelines that read and write from/to the feature store. This enabled a unified architecture, that we call the feature, training, inference (FTI) pipeline architecture, to describe interactive, batch, and streaming ML systems. In the FTI architecture, see figure 1, a ML system is composed of three different independent ML pipelines that share data by reading and writing to a common shared data layer - the feature store and a model registry. These ML pipelines have well-defined inputs and outputs can can be independently developed and operated:

- a *feature pipeline* transforms input data into features that are stored in the feature store;
- a *training pipeline* reads features and labels from the feature store, trains a model, and outputs the trained model to a model registry
- an *inference pipeline* reads new feature data and an ML model as input and produces both predictions and prediction logs as output.

Hopsworks supports many different types of **feature pipelines** - batch programs in Python, Spark, SQL, and streaming pipelines in Flink, Spark Streaming, and Beam. Feature pipelines typically do not need specialized hardware to run, such as a GPU. The choice of data processing framework is typically based on feature freshness (how old can the precomputed feature data be that is made available to models) and data volume requirements. Real-time ML systems often use stream processing, while most other ML systems use batch pipelines.

Training pipelines are typically Python programs that read a consistent snapshot of training data from the feature store as input, train a ML model (sometimes using hardware acceleration), evaluate/validate the model, and then store the trained model in a model registry.

Inference pipelines make predictions using the packaged model, downloaded from the model registry, and features read from the feature store and/or computed from user input data. Inference pipelines are typically implemented in Python, and when batch inference pipelines need to process large volumes of inference data, PySpark is often used.

2.2 Data Transformation Taxonomy

Data transformations create features from input data, but not all classes of transformation can or should be applied in all of the feature, training, and inference pipelines, see figure 1. We designed a taxonomy that identifies three different types of data transformation and in which ML pipelines those transformations can be applied:

- *model-independent transformations* convert input data into one or more reusable features/labels that can subsequently by used by one or more models;
- *model-dependent transformations* convert feature(s)/label(s) into one or more encoded/scaled features/labels for use by a single model;
- *on-demand transformations* convert request-time data (optionally along with other parameters) into one or more reusable features/labels for online models.

Model-independent transformations are implemented in either batch or streaming feature pipelines. Examples of model-independent data transformations include aggregations (windowed counts/sum, avg, max, min, etc), embeddings, and binning. Text chunking for large language models (LLMs) is also model-independent, as different LLMs and retrieval augmented generation (RAG) stores can reuse the text chunks. If the data volume is small, Python frameworks such as Pandas or Polars are popular choices, but for data volumes that don't fit on a single machine, PySpark and Data Warehouses (DBT/SQL) is popular for batch processing. For stream processing, Flink, Beam, and Spark Streaming are widely used. The choice of whether to implement a feature pipeline as a batch or streaming program depends on the feature freshness requirements - if the features need to be available for query in near real-time, stream processing is needed. Model-independent transformations are only applied in feature pipelines.

Model-dependent transformations are data transformations that are specific to one model such as feature encoding/scaling. They need to be applied in both training and batch/online inference pipelines, and as training and inference pipelines are separate

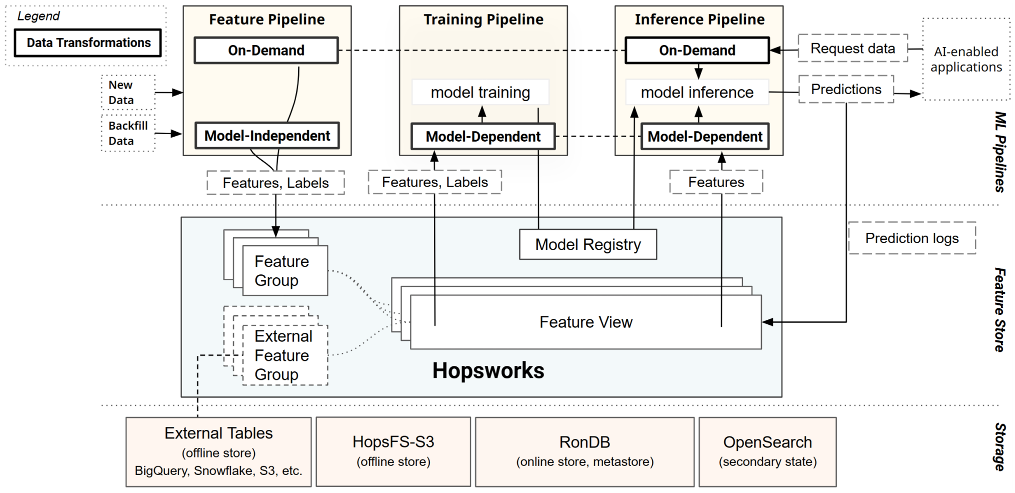


Figure 1: ML Systems based on a feature store implement 3 ML pipelines: a feature pipeline, a training pipeline, and an inference pipeline.

systems, there should be no offline/online skew between the data transformations in training and inference. An example of a model-dependent transformation is normalizing a numerical feature, as is often done in gradient-descent based model for performance reasons. However, the same numerical feature should not be normalized for a decision tree model. Hence, normalization (and all feature encoding/scaling, in general) is specific to one model. Model-dependent transformations are also easily identified as they are parameterized by the training dataset - before you can encode/scale a feature, you first need a full pass of the training data, for example, to compute the mean and standard deviation so that you can normalize all feature values. Model-dependent transformations should not be performed in a feature pipeline as they lead to *write amplification* - every write to a feature store requires the existing feature values to also be rewritten, as the new data written may have changed the mean/min/max/standard-deviation of a feature, so all feature values now need to be transformed. In any case, exploratory analysis of feature data is an important capability of feature stores, and storing encoded feature data makes this extremely difficult for data scientists. Another example of a model-dependent transformation is text tokenization for a LLM, as each LLM has their own tokenization algorithm. Solutions to prevent training/serving skew for model-dependent transformations include performing them in pre-processing pipelines (Scikit-Learn, Keras, or PyTorch) or using declarative transformations on a feature view, introduced in the feature view section.

On-demand transformations are performed in online inference pipelines and require request-time data to be computed (they can also use other external or historical data as input parameters). The same on-demand transformation can also be performed in a feature pipeline to process historical data and create resuable features. In contrast to model-dependent transformations, on-demand transformations are not parameterized by the training dataset. As online inference environments are almost exclusively Python environments, in Hopsworks, on-demand transformations are Python functions - either user-defined functions (UDFs) or more commonly Pandas UDFs. Pandas UDFs are preferred as they support vectorized operations and can be scaled out in Spark feature pipelines. On-demand transformations add some latency to online inference pipelines, and vectorized Pandas UDFs help reduce online model prediction latency when input data is large, compared to Python UDFs. As on-demand transformations are applied in both feature and online inference pipelines, they need to be consistent to prevent offline/online skew. One solution is to use versioned source-code control or a versioned Python package containing the UDFs in modules. Another solution is to have the online inference pipeline download the Python package using lineage information to go from the model to the feature to the feature pipeline module containing the UDF(s).

3 FEATURE GROUPS

In Hopsworks, features are computed in feature pipelines and stored together in mutable tables of related precomputed features called feature groups, see figure 1. The choice of which features to include in which feature group is informed by the data model and the update cadence for the data sources used to compute the features. A feature pipeline can update one or more feature groups.

The feature group itself is a *schema*, *metadata*, and *tables* in the offline and online stores. The **schema** can be provided explicitly or implicitly via a *DataFrame*. The feature group **metadata** contains a user-provided name, a version, a primary key, and a flag specifying whether it should be *online-enabled* or not. The primary key is needed to retrieve rows of online feature data and prevent duplicate data, while the version number enables support for A/B tests of features by different models and ML system upgrades. Additionally, the feature group metadata can also include a partition key, for partition pruning queries to the offline store, lineage information (parents/children of the feature group), and any embedding features that should be indexed for similarity search. The feature group backing **tables** are stored in RonDB (the online store), and the offline store can be either an external table in a data warehouse or an Apache Hudi copy-on-write table, stored on HopsFS/S3.

Our online store, RonDB, is a distributed, shared-nothing, highly-available database that supports both in-memory and on-disk data. RonDB supports transactions using a non-blocking two-phase commit (2PC) protocol that includes a third phase, a complete operation, with transaction deadlock/failure detection timeouts of just a couple of seconds to ensure real-time failover [41]. RonDB is a fork of MySQL Cluster (NDB), and it adds cloud-native capabilities, such as elastic scaling, availability-zone aware replication, and cloud-native backups. RonDB also supports asynchronous region-level replication in either active-standby mode or active-active mode with collision detection and avoidance support.

For online-enabled feature groups, a table is created in both RonDB and the offline store. For offline-only feature groups, only the Hudi table is created. For external feature groups, the table needs to already exist in the external data warehouse before the feature group is created. Hopsworks also includes a vector database (OpenSearch [33]) to index embedding features. Indexed embeddings enable the search for similar rows in the feature group using approximate nearest neighbor (ANN) search (faiss or nmslib) [37]. If a feature group contains an indexed embedding, an index is created (or re-used) in OpenSearch.

Append/upsert/delete data operations to/from a feature group can be performed using either the batch API or stream API. The batch API supports writing Spark DataFrames, and the stream API supports both Spark or Pandas DataFrames as well as Flink/Beam Datastreams. The batch API is only supported for offline feature groups, while all writes to online-enabled feature groups are via the stream API, which transparently writes data to both the online and offline tables. For offline-only feature groups, data is written to Apache Hudi, an open table format (like Apache Iceberg [22] and Delta Lake [2]) that stores tabular data as files in low-cost object stores, but also provides ACID updates, time-travel, primary key indexes, clustering indexes, and data skipping indexes. Hopsworks adds missing database management capabilities on top of Apache Hudi, such as a query service, access control, storage management, versioning, and background housekeeping tasks.

```
df = # Spark/Pandas DataFrame from live/historic data
# Perform feature engineering on 'df'

expectation_suite.add_expectation(
      ExpectationConfiguration(
    expectation_type="expect_column_values_to_be_in_set",
    kwargs={ "column":"fraud_label", "value_set": [0,1]})
)

fg1 = featurestore.create_feature_group("name",
    version=1, primary_key=["id"], event_time="ts",
    partition_key=["month"], online_enabled=True,
    expectation_suite=expectation_suite)
fg1.insert(df)
```

Listing 1: Upsert features to a feature group

In listing 1, we see how a feature group is created and populated with a DataFrame. By parameterizing how data is read from the source into the DataFrame, this minimal feature pipeline can be run with either new data as input or with historical data by parameterizing it with a start time and end time for the source data in a process known as backfilling. A feature pipeline can be run on a schedule (batch), potentially as part of a directed acyclic graph of jobs, or continuously (streaming). In this example, the feature group is *online_enabled*, so our client writes to the stream API. Writes to a feature group's stream API publish the data to a topic in Kafka with the same schema as the feature group. Hopsworks manages the lifecycle of the Kafka topic transparently, creating and destroying the topic along with the feature group tables, see figure 2. Hopsworks also supports a generic schema-less Kafka topic shared among many feature groups, which is useful for Enterprises where the ability to create/delete Kafka topics is restricted.

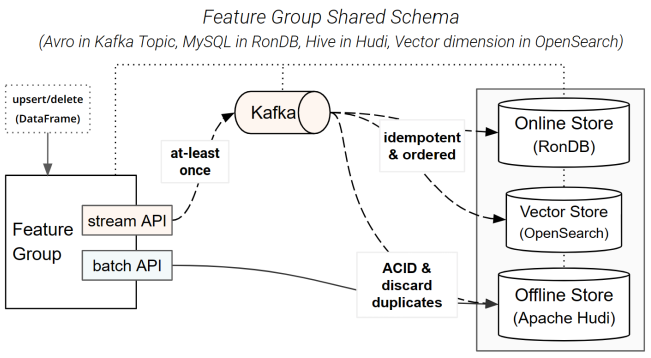

Feature Group Shared Schema
(Avro in Kafka Topic, MySQL in RonDB, Hive in Hudi, Vector dimension in OpenSearch)
upsert/delete (DataFrame)
Feature Group
stream API
batch API
at-least once
Kafka
idempotent & ordered
ACID & discard duplicates
Online Store (RonDB)
Vector Store (OpenSearch)
Offline Store (Apache Hudi)

Figure 2: Feature groups can be written to via a batch API (offline only) or stream API (online and offline). For writes via the stream API, Hopsworks guarantees eventually consistent updates to both the offline and online tables

From the Kafka topic, Hopsworks ensures eventually consistent replication of the data to both the offline and online stores, preventing *skew* between the data. Kafka provides at-least-once guarantees for data written to it, and Hopsworks has a connector service that copies the data from Kafka to RonDB (and OpenSearch), ensuring no duplicates in the online store by performing idempotent updates to feature values with their primary key. If there are multiple concurrent applications writing to an online feature group, it is possible for updates to arrive out-of-order, but, in practice, online feature groups are updated by a single client (a large Spark or Flink job is a also single client). For writes from Kafka to the offline store, we run a Spark job, called Hudi Delta Streamer, which ensures ACID updates and removes duplicate rows using Hudi's global index. Hudi Delta Streamer jobs can be run eagerly after a client finishes writing a DataFrame to Kafka or lazily to batch updates for better resource utilization.

Hudi tables are stored on HopsFS-S3, which provides tiered storage for files (small files in RonDB, recent file blocks on NVMe disk, and all files S3), proving faster query performance for recently accessed data. HopsFS can store small files in RonDB [32] and also implements a global, network-aware cache for recently accessed file blocks, stored on worker nodes in local NVMe disks. Writes to Hudi tables are via HopsFS' HDFS API and writes are both pushed to both the S3-compatible object store for persistent storage and to the local NVMe disks. HopsFS is highly available as it consists of redundant name nodes (metadata servers), redundant data nodes, and all of its metadata and small files are stored in RonBD, which is also highly available.

3.1 Extended Metadata for Feature Groups

You can design and attach schematized tags to feature groups (and other ML assets) in Hopsworks. This enables the design of data governance rules (e.g., data retention policy/date, personally identifiable information, scope-of-use for features) that can be uniformly applied across a feature store. All custom metadata for ML assets, as well as name/version/description information, is indexed in OpenSearch, enabling free-text search. You can also define data validation rules, using Great Expectations [14], that are attached as metadata to feature groups that are evaluated when you upsert data to a feature group. For example, you could define as expectations the valid range for a numerical feature or the valid set of categories for a categorical value, and those expectations will be validated before writing to the feature group. You define the policy for failed data validation rules - either fail the write or allow the write to proceed, but log a warning and/or trigger an alert by email/slack/etc.

3.2 Data Models for Feature Groups

Each feature group can have two different tables - one in the offline store and one in the online store share, and those tables will share the same data model. However, data warehouses (offline) and row-oriented databases (online) often have different preferred data models. Popular data models for data warehouses include one big table (OBT), fact-dimension data models (star schema or snowflake schema), and data vaults (with hubs, links, and satellites). OBT is not an efficient data model for online feature groups, as the data duplication introduced by denormalized tables is expensive for row-oriented databases (columnar compression is not possible). For this reason, row-oriented databases often use normalized data models that minimize duplicate data. In general, the preferred data model for feature groups is a normalized data model. The fact-dimension and data vault data models fit well with feature stores, as facts/satellites are typically the labels, and dimensions/hubs are the features. The foreign keys linking labels and features are either found in the fact/dimension tables or in the link tables for data vaults. When an online client wants to retrieve precomputed features, it needs to provide the foreign keys (as there is no label at prediction time). For this reason, the snowflake schema is a good data model, as it minimizes the number of foreign keys online clients have to provide when retrieving precomputed features - foreign keys can be provided by dimension tables. For example, if our label is in an order table and it has a foreign key to a product table, which in turn has foreign keys to factory and supplier tables, the client only needs to provide the primary key to the product table to be able to retrieve features from all 3 dimension tables. With a star schema, the client needs to provide 3 primary keys to all 3 dimension tables. Feature groups also tend to be wide tables, sometimes containing hundreds or more of features. As features can be reused across many models, projection pushdown is important in the online store to reduce the latency of feature lookups for online models. If projection pushdown is not available, feature groups containing features that will be reused should not contain too many columns, as this will increase read latency from the online store.

4 FEATURE VIEWS

Feature groups store feature data written by feature pipelines, but when we read feature data for training/inference, we often need to query across many feature groups. To complicate matters, feature groups store time series data, with each row having a timestamp column that indicates when the labels or feature values were observed. However, the feature pipelines that update the feature groups typically run at different cadences, so the timestamps of the feature groups don't align. This means that creating training data from feature groups requires a temporal Join to retrieve the feature values AsOf the timestamp for the label value, thus preventing data leakage in the training data. However, many data warehouses and data processing engines, including Spark, do not support temporal joins, which leads to complex, error-prone join queries built using something like a window function and an inequality join [11, 34].

We introduced the feature view as an abstraction to solve the following problems related to reading data from feature stores: (1) provide a simple Python/Spark API for reading point-in-time correct feature/label data for training and inference, (2) provide a single model schema to prevent incompatibility between training and inference pipelines, (3) enable reproduction of training data using only metadata, and (4) provide declarative support for model-dependent transformations of features. A feature view is metadata that represents the schema for a model (input features and output label(s)), filters applied when reading data, and any other helper columns needed to assist in training or inference (to compute on-demand features) or to store/process predictions.

A feature view is created when you want to create a model with a new schema. Creating a feature view starts by selecting its features, labels, and helper columns. These can all potentially come from different feature groups, as long as there is a foreign key joining the feature groups. A data model such as the fact-dimension model enables the joining of features to a label feature group. The data vault model or a snowflake schema additionally enables the transitive joining of features in linked feature groups.

To address the first problem, Hopsworks provides a simple Pandas-like domain-specific language in Python/PySpark for joining features to create a feature view, which can later be used to read point-in-time correct features and labels for training and inference.

```python
fg1 = featurestore.get_feature_group(fg1_name, version=1)
fg2 = featurestore.get_feature_group(fg2_name, version=1)

# Select and join features from different feature groups
selection = fg1.select_all()
               .join(fg2.select(["f1", "f2"]), on="id")

fv = featurestore.create_feature_view(
    "name", version=1, query=selection, label=["target"],
    transformation_functions={"f1": standard_scaler})

X_train, X_test, y_train, y_test = fv.train_test_split(
    test_ratio=0.2, extra_filter=fg2.region=="US")

model = XGBClassifier()
model.fit(X_train, y_train)
# Now register model with model registry
```

Listing 2: Python API for reading point-in-time correct feature/label data for model training

In listing 2, a feature view is created and used to create point-in-time consistent training data, with a random split (80/20) for the train and test sets (time-series splits are also supported). There is an additional filter on the partition key *region* for feature group *fg2* that prunes parquet files with records not in the 'US' region. The *train_test_split* method call is transpiled by Hopsworks into a SQL query, with help from Apache Calcite, that is then executed by our query service, and the result is returned as a Pandas DataFrame.

For inference, a feature view can also be used to read data for batch inference and online inference. Batch inference data is data that arrived in a recent time window (e.g., yesterday's data) or data retrieved by providing a spine feature group as a parameter that contains the IDs and timestamps for the feature values to be read. Online inference data contains precomputed feature values used in feature vectors for online models. Moreover, feature views allow for reading helper columns along with training, batch inference, or online inference data. Helper columns can be used to optimize training (e.g., active learning), to include information about where or how to store predictions, or historical feature values used to compute on-demand features.

The second problem addressed by feature views is that it prevents engineering mistakes when updating the model schema, shared between training and online inference pipelines. As training and online inference pipelines are different programs with different source code, model schema inconsistencies could arise if an update in one system was not consistently applied to the other system. Feature views prevent schema inconsistencies, helping eliminate a potential source of error in dynamically typed Python programs.

The third issue addressed is the reproduction of training datasets exclusively from their metadata. Hopsworks stores metadata for training datasets, including the splits, filter values, and other parameters, enabling their re-creation with an API call or in the UI.

Training datasets also store descriptive statistics for their features as metadata in Hopsworks. Those statistics help solve the fourth problem addressed by model-dependent transformations for feature views. Model-dependent transformations (such as feature encoding/scaling) can be declaratively attached to features in a feature view, and when a client retrieves feature data, the transformations will be executed in the Python client as a UDF using the training data statistics. Transformations such as encoding categorical features or normalizing numerical features require training data statistics and metadata. Reading data with a feature view, whether in training or inference, ensures the transformations are consistently applied using the same training dataset metadata.

4.1 Feature Reuse means Joins

Feature views capture the differences between reading training data versus reading inference data. In training, the label feature group is available as the labels are needed for supervised learning, but the label feature group is not available at inference time (your model predicts the label values). Similarly, on-demand features can be backfilled with historical data and are available when reading training data. However, during online inference, on-demand features need to be computed from request-time parameters and then merged with precomputed features read from the online store. As such, SQL queries used to read training data are different from those used to read inference data.

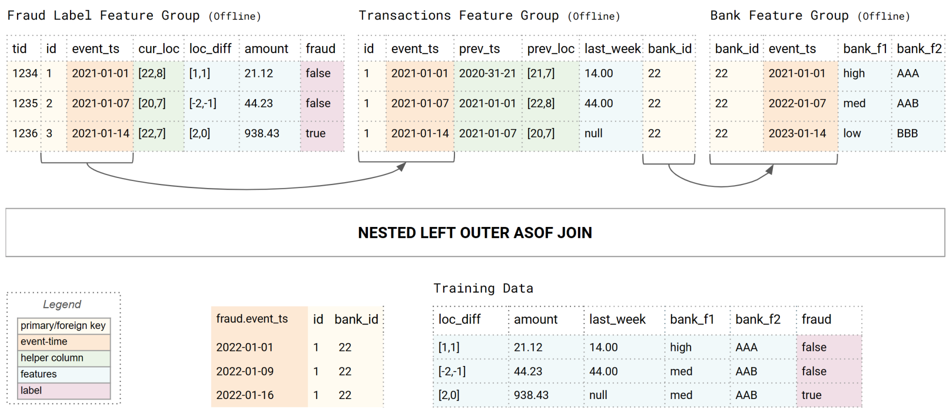
We use a AsOf Left join to read labels and features as training data from different feature groups. Transitive Left joins in a snowflake schema are also supported. The label feature group has a timestamp column (*event_ts).* For each row in the label feature group, we join feature values from different feature groups AsOf the *event_ts* in the label feature group. The AsOf Left join can nest if a feature group was joined to another feature group during feature selection for the feature view. We use a Left join, because we want to include all rows from our label feature group as training data. If a feature value is missing for a given *event_ts*, we still include the row in our training data with nulls for missing features. Later in a model-dependent transformation, a value can be imputed for the missing feature value, consistently between training and inference.

Figure 3 shows an example, from credit card fraud prediction, of a nested AsOf Left join used to create training data. Notice the *cur_loc*, *prev_ts*, and *prev_loc* columns are not included in the bottom table. Those columns are used in the feature pipeline as parameters to an on-demand feature function (a UDF) that computes *loc_diff* and stores it in the fraud label feature group - *loc_diff* is the location difference feature we use to help identify if two consecutive transactions have happened unrealistically far apart in too short a period of time.

4.2 Building Feature Vectors for Online Models

In inference, we want to predict the label using a trained model and feature values as input. These feature values can be a combination of precomputed features, on-demand features, and features *passed* as part of the prediction request. A client retrieves precomputed features for a model using the foreign keys from the label feature group. For nested feature groups in a feature view, the foreign keys are resolved as part of the query. If there are many foreign keys in the label feature group, our client issues parallel queries, as all the queries are independent of one another, finally merging the results.

Figure 4 shows how the feature vector is constructed from a prediction request using a combination of precomputed features and on-demand features. The code for figure 4 is shown in listing 3. The example code is a credit card fraud detection system that includes as prediction request parameters (1) the credit card number (*id*), the time it was used at (*event_ts*), the amount of money spent (*amount)* and the location (*cur_loc*) where the transaction took place. The feature store client uses the *id* to retrieve the precomputed features from the feature view (2) that pushes down a Left join to data nodes in RonDB. The *id* column is the foreign key to the *Transactions* feature group,and the *bank_id* is joined as part of a nested query that also returns features from the *Bank* feature group. The query retrieves both the precomputed features and helper features (*prev_ts*, and *prev_loc*), and merges them with *amount* (a feature passed as a runtime parameter). The helper features, along with two request parameters (*event_ts*, and *cur_loc*), are passed to an on-demand feature function that computes the *loc_diff* feature. The on-demand feature, *loc_diff,* is joined with the 3 precomputed features, and a feature passed as part of the prediction request, *amount*, to make up the untransformed feature vector, *merged*, in the client. In this example, the model is part of a scikit-learn pipeline


Fraud Label Feature Group (Offline)
Transactions Feature Group (Offline)
Bank Feature Group (Offline)
NESTED LEFT OUTER ASOF JOIN
Legend
primary/foreign key
event-time
helper column
features
label
Training Data

Fraud Label Feature Group (Offline)

tid	id	event_ts	cur_loc	loc_diff	amount	fraud
1234	1	2021-01-01	[22,8]	[1,1]	21.12	false
1235	2	2021-01-07	[20,7]	[-2,-1]	44.23	false
1236	3	2021-01-14	[22,7]	[2,0]	938.43	true

Transactions Feature Group (Offline)

id	event_ts	prev_ts	prev_loc	last_week	bank_id
1	2021-01-01	2020-31-21	[21,7]	14.00	22
1	2021-01-07	2021-01-01	[22,8]	44.00	22
1	2021-01-14	2021-01-07	[20,7]	null	22

Bank Feature Group (Offline)

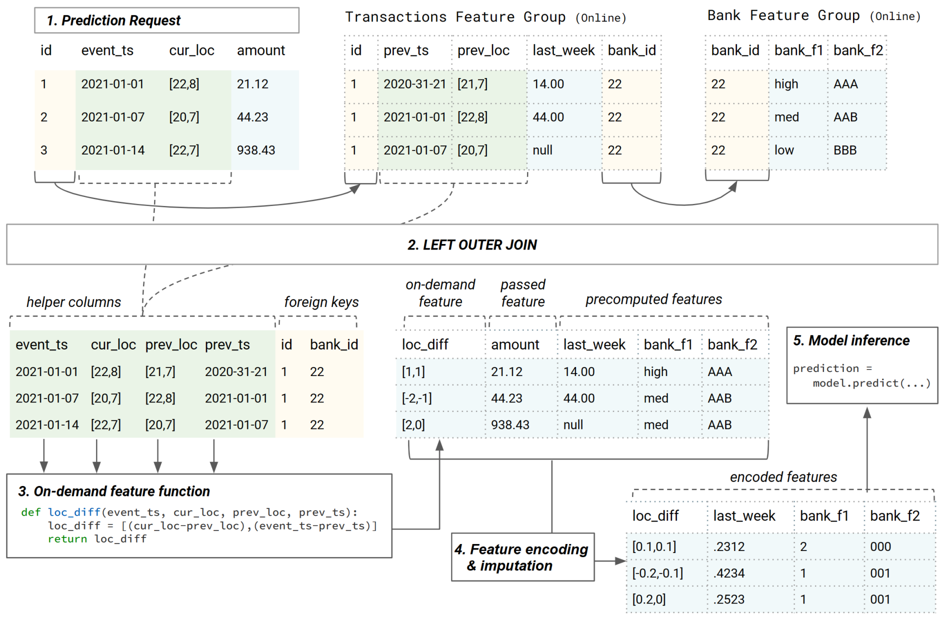
bank_id	event_ts	bank_f1	bank_f2
22	2021-01-01	high	AAA
22	2022-01-07	med	AAB
22	2023-01-14	low	BBB

fraud.event_ts	id	bank_id
2022-01-01	1	22
2022-01-09	1	22
2022-01-16	1	22

Training Data

loc_diff	amount	last_week	bank_f1	bank_f2	fraud
[1,1]	21.12	14.00	high	AAA	false
[-2,-1]	44.23	44.00	med	AAB	false
[2,0]	938.43	null	med	AAB	true

Figure 3: Create training data by starting from the label feature group and joining columns from other feature groups using (potentially nested) AsOf Left joins.



1. Prediction Request

id	event_ts	cur_loc	amount
1	2021-01-01	[22,8]	21.12
2	2021-01-07	[20,7]	44.23
3	2021-01-14	[22,7]	938.43

Transactions Feature Group (Online)

id	prev_ts	prev_loc	last_week	bank_id
1	2020-31-21	[21,7]	14.00	22
1	2021-01-01	[22,8]	44.00	22
1	2021-01-07	[20,7]	null	22

Bank Feature Group (Online)

bank_id	bank_f1	bank_f2
22	high	AAA
22	med	AAB
22	low	BBB

event_ts	cur_loc	prev_loc	prev_ts	id	bank_id
2021-01-01	[22,8]	[21,7]	2020-31-21	1	22
2021-01-07	[20,7]	[22,8]	2021-01-01	1	22
2021-01-14	[22,7]	[20,7]	2021-01-07	1	22

loc_diff	amount	last_week	bank_f1	bank_f2
[1,1]	21.12	14.00	high	AAA
[-2,-1]	44.23	44.00	med	AAB
[2,0]	938.43	null	med	AAB

```
def loc_diff(event_ts, cur_loc, prev_loc, prev_ts):
    loc_diff = [(cur_loc-prev_loc),(event_ts-prev_ts)]
    return loc_diff
```

```
prediction =
    model.predict(...)
```

loc_diff	last_week	bank_f1	bank_f2
[0.1,0.1]	.2312	2	000
[-0.2,-0.1]	.4234	1	001
[0.2,0]	.2523	1	001

Figure 4: Online inference starts with a prediction request (1) which includes a foreign key (id), 2 helper columns and a feature. The *id* is used to retrieve the precomputed features and helper columns from a feature view that spans 2 feature groups (2). The helper columns are now used to compute an on-demand feature (3), and then (4) the feature vector is encoded and missing values imputed (model-dependent transformations) before the model calls predict on it.

that first performs the model-dependent transformations (encoding, imputation) before the prediction is performed on the model. The prediction is then returned along with the untransformed features. The prediction is sent to the client, while the prediction logs are stored along with the prediction in the feature store.

```
def predict(id, event_ts, cur_loc, amount):
    df = feature_view.get_feature_vector(entry=
        {"id":id}, passed_features={"amount":amount},
     inference_helpers=True)
    df2 = loc_diff(pd.Series([event_ts], pd.Series(
        [cur_loc]), df["prev_loc"], df["prev_ts"])
    merged = feature_view.merge(df, {"loc_diff":df2})
    return model.predict(merged), merged
```

Listing 3: Merge precomputed and passed features with an on-demand feature for online inference

5 FEATURE QUERY SERVICES

Given our query models for feature data, we developed two query services for the offline store - one for server-sized data and one for big data. We also introduced a pushdown Left join optimization for our online store, RonDB, along with our feature store client.

5.1 ArrowFlight/DuckDB and Spark Query Services

Hopsworks includes two query services for reading feature data and creating training data as files from the offline store. The first query service developed is based on Spark. This works well at large scale when creating training data as files, although Spark's lack of AsOf Left joins means that we had to develop a windowed implementation of the point-in-time (temporal) join. We implemented our own point-in-time join algorithms for Spark, including union, exploding, and early stop sort-merge join operators [34]. However, Spark is still an order of magnitude slower than our 2nd query service, based on DuckDB and Arrow Flight, that enables data scientists to work more interactively with the feature store, see figure 5.

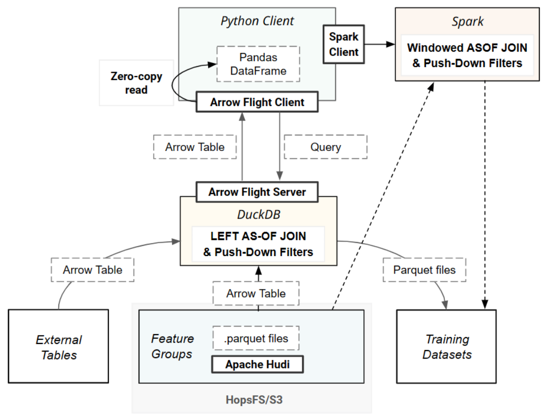


Figure 5: Offline Feature Query Service provides fast in-memory Pandas DataFrames via ArrowFlight, large training dataset creation via Spark.

Our second query service is an ArrowFlight Server that uses DuckDB to query the Parquet files in Apache Hudi tables, along with any external tables. For this, we developed a Hudi connector that returns the Parquet files for each feature group included in a feature view query. DuckDB can prune Parquet files with a partition key value, as Hudi supports Hive style partitioning, as well as push down filters in the feature view (or training data) query using statistics in Parquet files. DuckDB reads the (filtered) data from the Parquet files as Arrow tables, performing the AsOf Left joins directly on the Arrow tables, spilling to disk if needed.

Our server returns the Arrow data to the Pandas client using the Arrow Flight Protocol (Arrow is also an over-the-wire network format, not just an in-memory columnar data format). In contrast to JDBC/ODBC query services for feature stores (such as AWS Athena in Sagemaker), our service does not perform any column-to-row or row-to-column pivoting, and using Arrow means there is also no need to serialize/deserialize the data either from the Parquet files to the query service or from the query service to the Pandas/Polars clients. Pandas 2+ and Polars clients can read the arrow data directly into memory with zero copy semantics. Hopsworks also supports *federated queries* to external feature groups (tables). Our server first identifies the data source for the external feature group, then uses its connector to the data source (and any filters) to read the feature data as an Arrow table that is then passed to DuckDB for the temporal join.

5.2 Query Service for Online Features

Our online feature store is built on RonDB, a fork of NDBCluster [45] designed for cloud native operation that can store tabular data either in-memory or on-disk, see figure 6. RonDB includes a management node, that is also an arbitrator in the event of a split-brain network partition, data nodes, configured in replica groups typically of size 2 or 3, and API clients - either a MySQL Server, REST API, or native API clients (C++, Java). All nodes can be scaled horizontally and vertically. In this paper, we focus on the MySQL Server as a client API for the feature store, although the REST API server is also supported. The reason for this is that the SQL API has an additional pushdown optimization for Left joins that, as shown later, reduces latency and improves throughput.

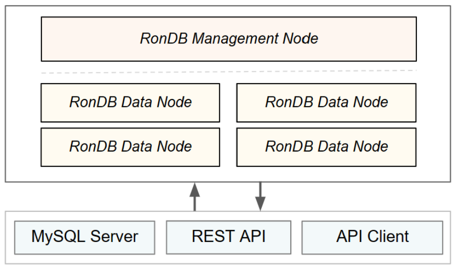


Figure 6: A RonDB Cluster

5.2.1 *Prediction Requests and the Data Model.*

The typical query workload for an online feature store is that feature store clients read the pre-computed features from the tables that make up a feature view using foreign keys to feature groups joined to the label feature group. In the case of a star schema data model for your feature view, you have N parallel primary key lookups for N joined feature groups. In the case of a normalized data model (e.g., snowflake schema or data vault), each foreign key can be either a primary key lookup, or a Left join (that potentially extends to multiple tables).

5.2.2 *Prediction Requests and the Feature Store Client.*

The feature store client issues parallel queries for all foreign keys that arrive in the prediction request. For each foreign key in the prediction request, we have an independent query, whether a primary key lookup or a Left join, and the Hopsworks feature store client issues these queries in parallel and merges the results to build the feature vector that is later sent to the model for prediction.

5.2.3 *Pushdown Optimizations for Feature Store Workloads.*

RonDB has several optimizations that improve its performance over key-value stores for online feature retrieval. Firstly, it supports predicate pushdown to read only those features needed from tables. This is important for popular features in wide tables (containing tens or hundreds of features), where lack of predicate pushdown means higher latency and excessive network bandwidth usage. Secondly, RonDB supports pushdown Left joins, which again helps reduce latency for feature lookups from a star schema (as seen in link tables in a data vault data model), see figure 7. Thirdly, RonDB supports pushdown nested Left joins, which again helps reduce latency for feature lookups in a snowflake schema data model, see figure 8. In both of these cases, the transaction coordinator (TC) handles the pushdown Left joins, reading and sending foreign keys to the participants (local data managers), that then send their results in parallel directly back to the client.

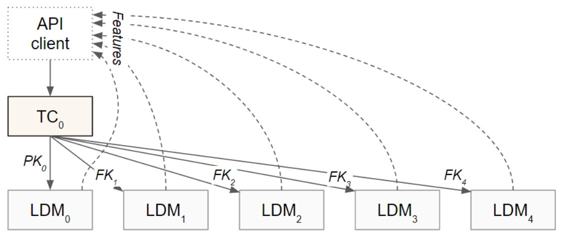


Figure 7: Pushdown Left (outer) join in RonDB for a star schema: the query is executed by the transaction coordinator (TC) that first reads the foreign keys from the first table using the primary key, and sends the reads with the foreign keys to the local data manager (LDM) threads in parallel at the (different) data nodes. Results are collected in parallel by the API client.

If features are heavily reused, across many models, there is a risk of hotspots in read access to RonDB. To overcome hotspots, RonDB supports fully-replicated tables, where rows are replicated to all database nodes in the cluster [43], increasing write overhead, but mitigating read hotspots. RonDB can also increase read throughput by increasing the number of read-only query threads on a database node through online elastic up-scaling of nodes [42]. Finally, RonDB can be replicated across data zones within a cloud region as well as across regions using geographical replication. RonDB supports Active-Active replication, with support for conflict-detection and conflict-resolution [44].

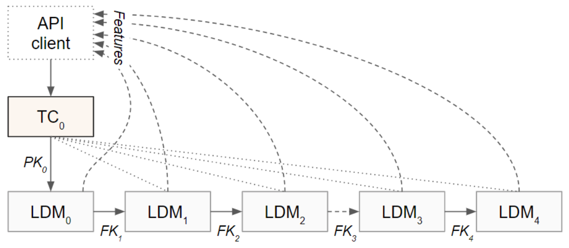


Figure 8: Pushdown Left join in RonDB for a snowflake schema: the query is executed by the transaction coordinator (TC) that reads the first foreign key from the first table using the primary key, then reads from the next table with the foreign key using the next local data manager (LDM) thread, that, in turn, reads the next foreign key from the next table, until all tables in the Left join are finished. Results are collected in parallel by the API client.

5.3 Similarity Search as a Query Service

Hopsworks also includes a vector database (OpenSearch kNN [37]), where features in a feature group can be indexed as embedding features for Approximate Nearest Neighbor (ANN) search [3]. When you define a feature group, you can specify which features are embeddings. It is the responsibility of the feature pipeline to use an embedding model to encode the higher dimensional data into embedding vectors. The vectors are copied to OpenSearch from Kafka using an eventually consistent replication protocol (see figure 2). Currently, we only index the latest feature values in OpenSearch, as this is required by the most common use case of a vector database - personalized recommendations using the ranking and retrieval architecture [5, 13, 46]. Hopsworks provides a Python API to define embedding features and query feature/label data using similarity search for both batch inference and online inference (see listing 4).

```python
from hsfs.embedding import EmbeddingIndex,
      SimilarityFunctionType

index = EmbeddingIndex("product")
index.add_embedding(name="emb_feature", dimension=128,
     similarity_function_type=SimilarityFunctionType.L2)

fg = fs.create_feature_group(fg_name, version=version,
     primary_key=["id1"], event_time="ts",
     online_enabled=True, embedding_index=emb_index)

# Read feature/label data similar to a given embedding
similar_features = fg.find_neighbors(reference_emb)
# Retrieve feature vectors similar to a given embedding
feature_vectors = fv.find_neighbors(reference_emb)
```

Listing 4: Python API for reading feature/label data using similarity search and a given vector embedding

5.4 Prediction Logging and Feature Monitoring

Hopsworks supports logging predictions for both online inference and batch inference. Prediction logs are used for (1) debugging predictions by hand by inspecting individual feature values and outputs, (2) monitoring for feature drift, (3) evaluating model performance by comparing logged predictions with either outcomes (if available) or some proxy metric for model performance. It is important to log unencoded feature values and predictions, as monitoring in challenging for encoded data and debugging is almost impossible for humans with encoded feature data.

Monitoring of feature inference data allows for a faster identification and resolution of problems in a ML system [40, 47]. Issues can be introduced by code changes (e.g., data transformations) or data changes (e.g., statistically significant changes in the feature data compared to the data used to train the model). Data anomalies are propagated to dependent ML pipelines. For instance, changes in a feature pipeline can introduce anomalies in the produced feature values, whose statistical properties will diverge from those of the training datasets of production models. Monitoring features requires the computation of descriptive statistics on feature data over time, which can be event-based (e.g., new feature data written to the feature store) or on a schedule. Statistics can be computed on a subset of feature data using detection and reference windows, or on feature data being written to the feature store as part of the same commit. Visualizing statistics as a time series can help manually identify anomalies in feature values. Moreover, by configuring reference values to compare statistics with, and thresholds as an estimation for abnormal values, the monitoring of feature data can be made automatic and performed in the background.

As shown in listing 5, Hopsworks provides a simple but rich Python API to setup the monitoring of feature values over time, optionally defining detection and reference windows, and the comparison criteria to identify when there is a shift in feature data.

```
# Compute statistics on:
# - all features of a feature group per data ingestion
fg_mon = fg.create_statistics_monitoring("name").save()
# - on a specific feature and detection window
fg_mon = fg.create_feature_monitoring("name",
    feature_name="amount", job_frequency="WEEKLY")
.with_detection_window(row_percentage=0.8
    time_offset="1w") # fetch data from the last week
# Compare feature statistics with a reference window
fg_mon.with_reference_window(row_percentage=0.8
    time_offset="2w", # fetch data from the previous week
    window_length="1w")
# - or a reference value
fg_mon.with_reference_value(value=100)
# - or training dataset statistics
fg_mon.with_reference_training_dataset(version=1)
      .compare_on(metric="mean", threshold=50)
```

Listing 5: Monitor feature data for drift

6 AVAILABILITY ZONE AND REGION-LEVEL REPLICATION

Hopsworks can be configured to be highly available within a data center and also between data centers (cloud regions). Hopsworks has several internal stateful services that are replicated, including HopsFS, RonDB, Kafka, and OpenSearch. An external Kafka cluster can also be used, such as Confluent Kafka. Hopsworks online store has support for local reads within an availability zone, through the *LocationDomainId* variable in RonDB that identifies which availability zone a data node belongs to. RonDB clients can then provide their own *LocationDomainId* to preferentially start transactions within their own availability zone, helping reduce network traffic costs between availability zones. This is made possible as RonDB uses a non-blocking two-phase commit algorithm [45], so when a client reads from the local data node, the local data node will have a copy of the latest version of the data. In contrast, reads in a quorum-based replication protocols, used by most key-value stores, will always cross availability zones, assuming one replica per availability zone.

Hopsworks Feature Store can also be made highly available in a multi data center configuration, see figure 9. Hopsworks supports Active/Standby high availability across regions, where writes are performed on the Active cluster (mediated by an external arbitrator service), but reads can be performed on either cluster. RonDB asynchronously replicates online feature data and metadata between regions. Hopsworks builds on an Active-Active replicated Kafka cluster (such as Confluent Kafka [48]) that replicates writes across both data centers, as well as an Active-Active replicated

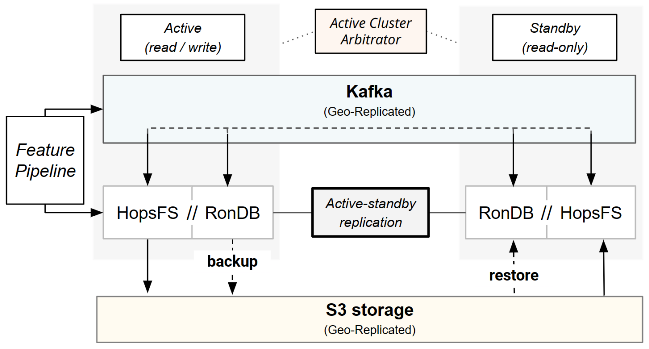


Figure 9: *Region-level replication of Hopsworks feature store*

S3-compatible storage (such as AWS S3 or Scality [19]), which replicates the files in the offline store between the Active and Standby clusters.

7 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of Hopsworks Feature Store for both offline and online workloads. There are new benchmarks for data and AI, such as TPCx-AI [6], that cover video, image, and text data and model training. However, feature stores manage primarily tabular data, so we restricted the benchmarks here to common workloads observed in the feature store community [17].

7.1 Offline Feature Store Benchmarks

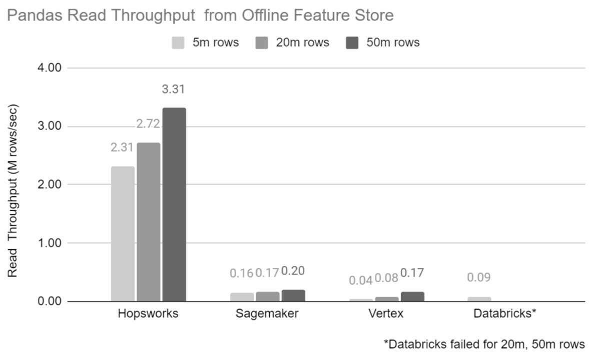
Given the lack of industry standard benchmarks for feature store workloads, we developed a benchmark [18] around the widely used NYC Taxi Dataset [36] to provide insights about read performance for the different offline feature stores for Pandas clients. We compare Hopsworks (ArrowFlight/DuckDB) with Sagemaker [4], Databricks [8], and Vertex [20], as these 3 feature stores are publicly available without the need to configure and attach a data warehouse, and they also have DeWitt Clause friendly licenses [18].

We run two separate experiments with batches of data of different sizes (5M, 20M, and 50M rows) - first reading from a single feature group, and then reading from 3 feature groups using a point-in-time join. The cluster details of the offline feature stores used in the benchmark are gathered in table 1, where all clusters ranged between 12-16 vCPUs and 48-64GB of memory. Note that Hopsworks Feature Store used NVMe disks.

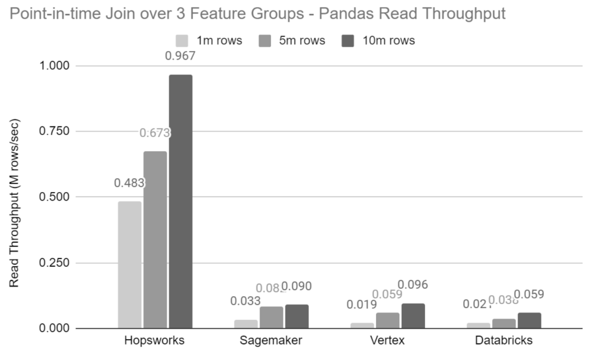
Feature Store	Cluster Details
Hopsworks	1 node, 16 vCPUs, 60GB, NVMe
Databricks	Driver, 3 workers (12 vCPUs, 48GB)
AWS Sagemaker	Notebook with 16 vCPUs, 64GB
GCP Vertex	Workbench with 16 vCPU, 60GB

Table 1: Offline Read Benchmark Cluster Setup

In figure 10(a), it is shown that, for a single feature group, Hopsworks, when reading 50M rows, reaches 16 and 19 times the throughput of Sagemaker and Vertex, respectively. In the case of Databricks, it was not possible to read 20M and 50M rows in the same read operation.



(a) Throughput for reading a Pandas DataFrame from a single Feature Group



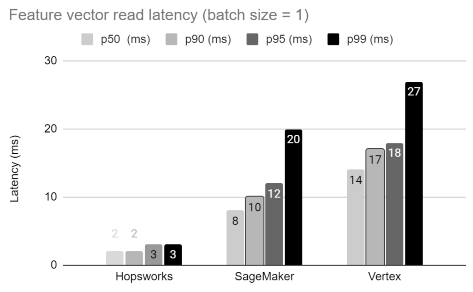
(b) Throughput for reading a Pandas DataFrame from 3 Feature Groups using a Point-in-time Join

Figure 10: Offline Store read throughput for Pandas clients.

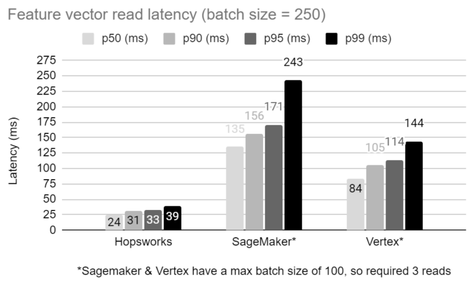
When reading from 3 feature groups using a point-in-time join (figure 10(b)), Hopsworks, when reading 10M rows, achieves 11, 10 and 17 times the throughput of Sagemaker, Vertex and Databricks, respectively. We only reached 10m rows, as we could not get Sagemaker, Vertex and Databricks to work at 50m rows.

7.2 Online Feature Store Benchmarks

We simulated two common experimental scenarios - lookups of individual feature vectors, and batch lookups, common in personalized recommendation systems, where 150-350 candidates often have their features retrieved in a single batch lookup from the feature store [5, 13, 46]. We ran locust [30] clients on client machines with 32 CPUs (c5a.8xlarge and e2-highcpu-32). Hopsworks/RonDB included 1 MySQL Server (t3.medium, 2 vCPU, 4GB) and 2 Data Nodes (m5.4xlarge, with 16 vCPUs and 128GB). GCP Vertex and AWS Sagemaker Online Feature Stores are managed services, where Sagemaker uses DynamoDB and Vertex does not divulge the database it uses (although, a forthcoming version has been announced that it is built on BigTable). We only evaluated latency, so our workloads did not saturate the databases. We didn't measure throughput as Vertex and Sagemaker are managed services with low throughput quotas. Databricks currently has no public API for their online store.



(a) Latencies for reading single feature vectors



(b) Latencies for reading batches of 250 feature vectors

Figure 11: Latency benchmarks for the Online Store.

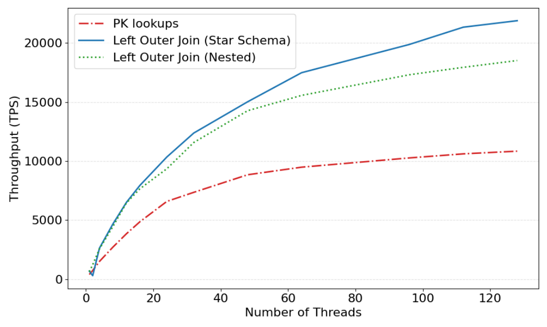
In figure 11, we can see that, for a single feature vector lookup, Hopsworks, for p99 latency, has 15% of the latency of Sagemaker and 11% of the latency of Vertex. Note that SageMaker and Vertex have a batch size limit of 100 records per request. Therefore, for batch size 250, we ran 3 sequential requests.

7.3 Pushdown Left Outer Joins in RonDB

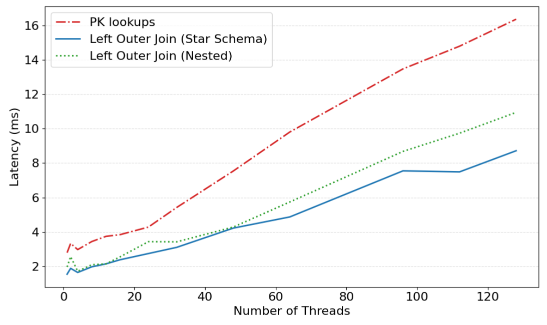
In this experiment, we show the benefit of pushdown Left (outer) joins compared to key-value lookups in RonDB. We present two different types of Left joins - a star schema model and a nested foreign key data model (snowflake model) to show high performance for even deeply nested tables. In the star schema model (e.g., the satellite table in a data vault model), all the 10 foreign keys are in the source table and link directly to the 10 different tables containing the features. In the nested snowflake scenario, all the features are retrieved from increasingly nested tables, linked from the starting table. There are 10 tables, each with a foreign key to another table, and you have resolve and traverse all 10 foreign keys to read all the features in these 10 feature tables. For this nested Left join, we show the performance for an equivalent key-value lookup implementation, where we perform 10 sequential PK lookups, approximating the performance of a key-value store compared our pushdown Left joins.

For the hardware used in this experiment, all the VMs are c3d VM types in GCP. There was 1 RonDB Data Node with 4 vCPUs. There were 2 MySQL Servers with 8 vCPUs each. There was 1 API node running the benchmark program. The test used was Sysbench OLTP ReadOnly [9] with all range queries set to 0. For the PK lookups scenario, the transaction included 10 PK lookups. For the

Left join (star schema), one query was executed per transaction with 10 tables, where each table was accessed through the PK. For the Left join (nested using foreign keys), one query was executed per transaction with 10 tables, where each table was accessed through a foreign key to the next table. The Left join queries are a new add-on to the sysbench-0.4.12 that is integrated with RonDB releases.



(a) Throughput for Primary Key Read, Left Joins (Star Schema and Nested)



(b) Latency for Primary Key Read, Left Joins (Star Schema and Nested)

Figure 12: Throughput and Latency for Primary Key Lookups and Left Joins

In figure 12, we can see that the pushdown Left join in RonDB results in latency and throughput improvements versus iterative primary key lookups, for both the star schema data model and a nested (foreign keys) data model. For latency, in the star schema model, we see a 35-50% reduction in latency and a 20-43% reduction for the nested data model. For throughput, in the star schema model, we see a 57-102% increase in throughput and a 42-71% increase for the nested data model.

8 RELATED WORK

Feature stores are now widely used as a data management layer for ML systems. Prior to Hopsworks, the first feature stores, Michelangelo Palette by Uber and Zipline by Airbnb [12], were based on a domain-specific language (DSL) for creating feature pipelines that write features to the offline and online stores. Both these DSLs compile into Spark jobs that compute features that are then written to the online and offline stores. In contrast, Hopsworks introduced the API-based feature store, as a DataFrame API for writing feature data. A number of open-source feature stores have been developed, including FeaSt [15], FeatureForm [16], and Feathr [29], as well as commercial feature stores, including Tecton [50], Databricks [8], AWS Sagemaker [4], and GCP Vertex [20]. Hopsworks is the only feature store that builds on our data transformation taxonomy with model-independent, model-dependent, and on-demand transformations. Other existing feature stores use key-value stores as online stores, and, therefore, do not support pushdown projections, or pushdown Left joins. For offline feature data, other feature stores use existing data warehouses that provide ODBC/JDBC APIs for retrieving feature data as Pandas DataFrames, and Hopsworks is the only feature store to provide an offline query service that uses AsOf Left joins (DuckDB) to implement point-in-time joins.

Temporal expressions were included in SQL:2011, including the AsOf join operator [26]. The goal of temporal data support was to define and associate time periods with the rows of a table, although there is no new datatype to represent a time-period data type, with a start/stop pair of datetime or timestamp values. In Hopsworks, we introduced the *event_ts* type, rather than a time-period, as an observation (label) is defined by a single timestamp, not a time-period. Features could, potentially, be defined as a period data type, but as labels can be features in one model, and vice-versa in another model, we chose to simplify the temporal data model and support a single event-time column. Our event-time is not the system-time from SQL:2011, but what is called the application-time. Another temporal data model that could be considered is Bitemporal tables that store both a system-versioned and an application-time period table - capturing both the periods during which facts were believed to be true in the real world as well as the periods during which those facts were recorded in the database. However, we believe the extra power of this temporal data model is not warranted, given the extra complexity it introduces. There are benchmarks that include temporal workloads, such as TPC-BiH [24]. Feature stores are most concerned with temporal joins, not temporal aggregations or time-travel queries, hence we didn't include them in our experiments.

Only a few columnar databases support AsOf joins including DuckDB [11], Clickhouse [7], QuestDB [38], KDB [27], and Kinetica [25]. The Python data processing library, Polars [35], also supports AsOf Left joins. Similar to our work on Spark point-in-time join optimizations [34], Featr [29] have also improved temporal joins in Spark showing a 3X improvement over baseline.

9 CONCLUSIONS

Hopsworks Feature Store is a DBMS for ML, that enables ML systems to be structured as independent feature, training, and inference pipelines. Hopsworks introduces a taxonomy for data transformations (reusable features, model-specific features, and on-demand features) and provides support for implementing those transformations as UDFs in the different ML pipelines. Hopsworks also introduced a query model for reading training data (AsOf Left joins) and online features (parallel Left joins), along with performance optimizations for reading feature data from the offline store and pushdown optimizations for the online store. The next steps for Hopsworks, and the feature store community in general, are to work further on integrating Python with Data Warehouses/Lakehouses and to support for real-time ML systems.

REFERENCES

[1] Hopsworks AB. 2020. *RonDB: a distribution of NDB Cluster.* Hopsworks AB. Retrieved Nov 24, 2023 from https://github.com/logicalclocks/rondb

[2] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał undefinedwitakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[3] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2020. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems* 87 (2020), 101374.

[4] AWS. 2020. Retrieved Nov 28, 2023 from https://aws.amazon.com/sagemaker/feature-store

[5] Paul Baltescu, Haoyu Chen, Nikil Pancha, Andrew Zhai, Jure Leskovec, and Charles Rosenberg. 2022. ItemSage: Learning Product Embeddings for Shopping Recommendations at Pinterest. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 2703–2711. https://doi.org/10.1145/3534678.3539170

[6] Christoph Brücke, Philipp Härtling, Rodrigo D Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI-An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3649–3661.

[7] Clickhouse. 2023. *Clickhouse Documentation.* Clickhouse. Retrieved Nov 24, 2023 from https://clickhouse.com/docs/en/sql-reference/statements/select/join#asof-join-usage

[8] Databricks. 2021. Retrieved Nov 28, 2023 from https://www.databricks.com/product/feature-store

[9] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[10] DuckDB. 2019. Introducing Apache Arrow Flight: A Framework for Fast Data Transport. Retrieved Nov 24, 2023 from https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight

[11] DuckDB. 2023. DuckDB's AsOf Joins: Fuzzy Temporal Lookups. Retrieved Nov 24, 2023 from https://duckdb.org/2023/09/15/asof-joins-fuzzy-temporal-lookups.html#what-is-an-asof-join

[12] Zanoyan et al at Airbnb. 2018. *Zipline Airbnb's ML Data Management Framework.* Airbnb. Retrieved Nov 24, 2023 from https://conferences.oreilly.com/strata/strata-ny-2018/public/schedule/detail/68114.html

[13] Ling et al at Uber. 2023. *Innovative Recommendation Applications Using Two Tower Embeddings at Uber.* Uber. Retrieved Nov 24, 2023 from https://www.uber.com/en-SE/blog/innovative-recommendation-applications-using-two-tower-embeddings/

[14] Great Expectations. 2017. Retrieved Nov 28, 2023 from https://greatexpectations.io

[15] Feast. 2019. . Feast. Retrieved Nov 24, 2023 from https://github.com/feast-dev/feast

[16] FeatureForm. 2019. The Open-Source Virtual Feature Store. Retrieved Nov 28, 2023 from https://www.featureform.com

[17] featurestore.org. 2020. Feature Store Summit 2023. Retrieved Nov 24, 2023 from https://www.featurestore.org/feature-store-summit-2023

[18] featurestore.org. 2023. *Feature Store Benchmarks.* featurestore.org. Retrieved Nov 24, 2023 from https://github.com/featurestoreorg/featurestore-benchmarks

[19] Frank Gadban and Julian Kunkel. 2021. Analyzing the Performance of the S3 Object Storage API for HPC Workloads. *Applied Sciences* 11, 18 (2021), 8540.

[20] Google. 2021. Retrieved Nov 28, 2023 from https://cloud.google.com/vertex-ai/docs/featurestore

[21] Apache Hudi. 2016. *Hudi Github Repository.* Apache. Retrieved Nov 24, 2023 from https://github.com/apache/hudi

[22] Apache Iceberg. 2017. . Apache. Retrieved Nov 24, 2023 from https://github.com/apache/iceberg

[23] Mahmoud Ismail, Salman Niazi, Gautier Berthou, Mikael Ronström, Seif Haridi, and Jim Dowling. 2020. HopsFS-S3: Extending Object Stores with POSIX-like Semantics and More (Industry Track). In *Proceedings of the 21st International Middleware Conference Industrial Track* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3429357.3430521

[24] Martin Kaufmann, Peter M Fischer, Norman May, Andreas Tonder, and Donald Kossmann. 2014. Tpc-bih: A benchmark for bitemporal databases. In *Performance Characterization and Benchmarking: 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers 5.* Springer, Frankfurt, Germany, 16–31.

[25] Kinetica. 2023. *Kinetica Documentation.* Kinetica. Retrieved Nov 24, 2023 from https://docs.kinetica.com/7.1/sql/query/#asof

[26] Krishna Kulkarni and Jan-Eike Michels. 2012. Temporal features in SQL: 2011. *ACM Sigmod Record* 41, 3 (2012), 34–43.

[27] KX. 2023. *KX Documentation.* KX. Retrieved Nov 24, 2023 from https://code.kx.com/q/ref/asof/

[28] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2483–2488. https://doi.org/10.1145/3514221.3522565

[29] Rui Liu, Kwanghyun Park, Fotis Psallidas, Xiaoyong Zhu, Jinghui Mo, Rathijit Sen, Matteo Interlandi, Konstantinos Karanasos, Yuanyuan Tian, and Jesús Camacho-Rodríguez. 2023. Optimizing Data Pipelines for Machine Learning in Feature Stores. *Proc. VLDB Endow.* 16, 13 (2023), 4230–4239. https://www.vldb.org/pvldb/vol16/p4230-camacho-rodriguez.pdf

[30] Locust.io. 2011. . locust.io. Retrieved Nov 24, 2023 from https://github.com/locustio/locust

[31] Igor L. Markov, Hanson Wang, Nitya S. Kasturi, Shaun Singh, Mia R. Garrard, Yin Huang, Sze Wai Celeste Yuen, Sarah Tran, Zehui Wang, Igor Glotov, Tanvi Gupta, Peng Chen, Boshuang Huang, Xiaowen Xie, Michael Belkin, Sal Uryasev, Sam Howie, Eytan Bakshy, and Norm Zhou. 2022. Looper: An End-to-End ML Platform for Product Decisions. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 3513–3523. https://doi.org/10.1145/3534678.3539059

[32] Salman Niazi, Mikael Ronström, Seif Haridi, and Jim Dowling. 2018. Size matters: Improving the performance of small files in hadoop. In *Proceedings of the 19th international middleware conference.* USENIX Association, USA, 26–39.

[33] OpenSearch. 2021. Using OpenSearch as a Vector Database. Retrieved Nov 24, 2023 from https://opensearch.org/platform/search/vector-database.html

[34] Axel Pettersson. 2022. *Resource-efficient and fast Point-in-Time joins for Apache Spark : Optimization of time travel operations for the creation of machine learning training datasets.* Master's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[35] Polars. 2023. *Polars Documentation.* Polars. Retrieved Nov 24, 2023 from https://pola-rs.github.io/polars/user-guide/transformations/joins/#left-join

[36] M Poongodi, Mohit Malviya, Chahat Kumar, Mounir Hamdi, V Vijayakumar, Jamel Nebhen, and Hasan Alyamani. 2022. New York City taxi trip duration prediction using MLP and XGBoost. *International Journal of System Assurance Engineering and Management* (2022), 1–12.

[37] OpenSearch Project. 2021. *OpenSearch k-NN.* AWS. Retrieved Nov 24, 2023 from https://github.com/opensearch-project/k-NN

[38] QuestDB. 2023. *QuestDB Documentation.* QuestDB. Retrieved Nov 24, 2023 from https://questdb.io/docs/reference/sql/join/#left-outer-join

[39] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[40] Stephan Rabanser, Stephan Günnemann, and Zachary Lipton. 2019. Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Neurips. https://proceedings.neurips.cc/paper_files/paper/2019/file/846c260d715e5b854ffad5f70a516c88-Paper.pdf

[41] RonDB. 2021. *Non-blocking 2PC.* Hopsworks AB. Retrieved Nov 24, 2023 from https://docs.rondb.com/rondb_nonblocking_2pc

[42] RonDB. 2021. *Query Threads in RonDB.* Hopsworks AB. Retrieved Nov 24, 2023 from https://www.rondb.com/post/rondb-automatic-thread-configuration

[43] RonDB. 2023. *Fully Replicated Tables in RonDB.* Hopsworks AB. Retrieved Nov 24, 2023 from https://docs.rondb.com/intro_relational/#fully-replicated-tables

[44] RonDB. 2023. *Geographic replication in RonDB.* Hopsworks AB. Retrieved Nov 24, 2023 from https://docs.rondb.com/rondb_multi_site/

[45] Mikael Ronström. 2018. *MySQL Cluster 7.5 inside and out.* BoD-Books on Demand, Stockholm, Sweden.

[46] Chuanwei Ruan, Allan Stewart, Han Li, Ryan Ye, David Vengerov, and Haixun Wang. 2023. Dynamic Embedding-Based Retrieval for Personalized Item Recommendations at Instacart. In *Companion Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 983–987. https://doi.org/10.1145/3543873.3587668

[47] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine Learning: The High Interest Credit Card of Technical Debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*. Google, Mountain View, CA, USA.

[48] Gwen Shapira, Todd Palino, Rajini Sivaram, and Krit Petty. 2021. *Kafka: the definitive guide.* " O'Reilly Media, Inc.".

[49] Splicemachine. 2012. Retrieved Sept 18, 2023 from https://www.splicemachine.com

[50] Tecton. 2019. Retrieved Nov 28, 2023 from https://www.tecton.ai

[51] Uber. 2017. Meet Michelangelo: Uber's Machine Learning Platform. Retrieved Nov 24, 2023 from https://www.uber.com/en-SE/blog/michelangelo-machine-learning-platform