Rice Simple Feature Store

COMP 536, Fall 2021, Chengzu Ou<co32>

OVERVIEW

A feature store is a centralized storage for the features used in machine learning training or inferencing. It is a relatively new concept proposed in 2017 by the machine learning infra team at Uber. It has become popular in the past two years. Many companies such as Uber, Airbnb, DoorDash, and Netflix have implemented feature stores for internal use. Cloud service providers such as Amazon, Google, and Databricks have launched managed feature store services. There are also some popular open-source projects such as Feast and Hopsworks. Someone in the community built this excellent website to help organize the recent development. We believe there will be a boom in machine learning platforms and tools in the next few years, and feature stores will be one of the hot areas.

PURPOSE

We will try to implement a simple but full-featured feature store for the course project to deepen our understanding. First, we want to capture the recent trends, techniques, and systems designing perspectives by analyzing currently available products in the industry. We would produce a short report hoping that it might be helpful to guide the design and implementation. Then, we would try to design a minimum viable feature store with just the most salient features based on the previous analysis. Then, we would build the system and deploy it. Finally, we would produce another document for the implementation details. We can try to iterate over the initial version if we have more time to make it robust.

SCOPE

To scope the project to a reasonable size to be done in one semester, we have to make some sacrifices. First, we would not put the performance in an important position, which means we would not expect it to be a super-fast, production-ready system. Second, we would focus on the top-level design, metadata management, and data flow. We will not try to reinvent bottom-level components such as the K-V store. Instead, we would go to the available open-source tools and cloud services whenever possible.

PRODUCTS ANALYSIS

We start by analyzing currently available feature stores on the market. By comparing the overall architecture and functionalities they offer, we can identify the most crucial features of a feature store, which will guide our design.

As we have mentioned above, there are many feature stores. Not only are startup companies proposing their feature store products, but cloud service providers have also started to build their own "feature store as a service." Based on the popularity, we picked four feature store products to analyze: Hopsworks Feature Store, Feast, Databricks Feature Store, and Sagemaker Feature Store. Two are open source, and two are from existing cloud service providers (Databricks and AWS).

Hopsworks Feature Store

The Hopsworks Feature Store is an open-source and managed service that connects to many data sources and supports feature computations in Spark or Python environments.

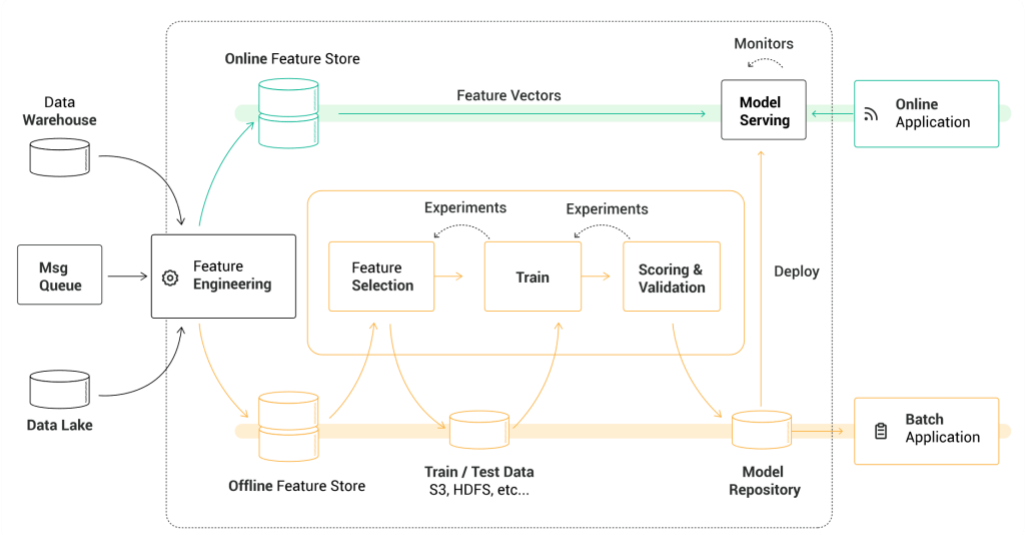

Figure 1 Hopswork Feature Store Architecture

Hopsworks online store is built on RonDB, a key-value store that Logical Clocks (the company behind Hopsworks) developed. It is a fast k-v store that can support "throughput of up to hundreds of millions of read/write ops/seconds and <1ms latency for individual operations." It also has SQL support which most of the k-v stores do not have.

For the offline store, Hopsworks uses HopsFS, an in-house distributed file system that is built on top of S3 with an HDFS API(POSIX-like) and is claimed to be 100x times faster than S3. It also supports open file formats like Parquet, Hudi, Delta Lake, and ORC.

Most feature stores provide a SQL API for the user to transform raw data into features (batch or streaming) and to retrieve features. Hopsworks is more Python-friendly, providing a Pandas-like API to better accommodate data scientists who are more comfortable using Python.

Besides, Hopsworks Feature Store also supports versioning bother feature schemas and feature values(time-travel).

Hopsworks is an open-source product backed by a commercial company (Logical Clocks). It has a rather built-on-their-own stack of technologies.

Feast

Feast is another open-source framework that enables users to access data from your machine learning models. It allows teams to register, ingest, serve, and monitor features in production.

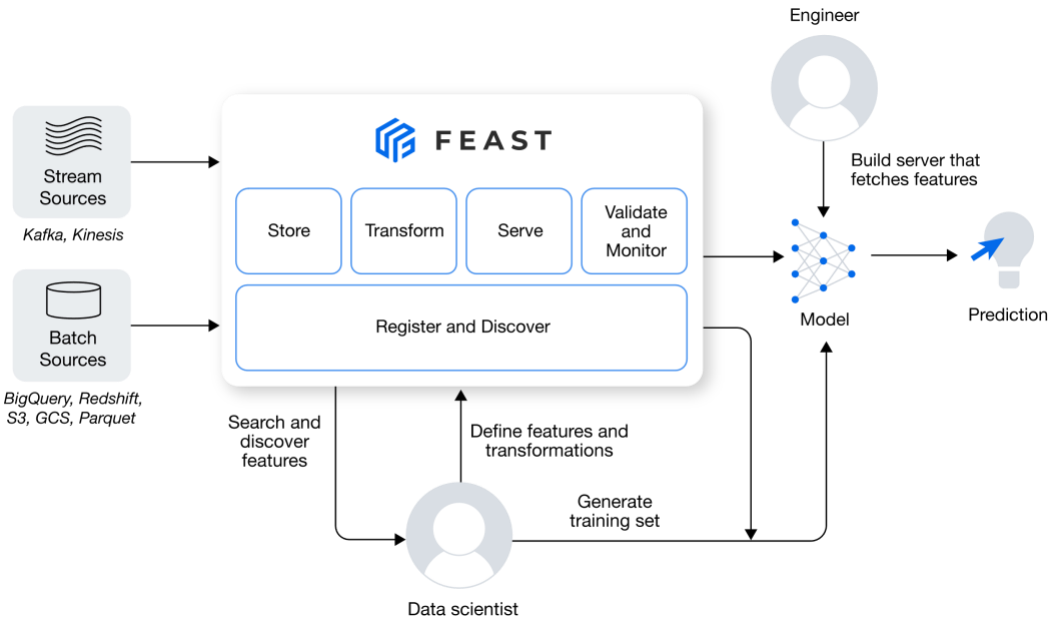

Figure 2 Feast Architecture

Rather than focusing on the implementation, Feast is more like a general framework for feature management. It uses ELT/ETL systems like Spark and SQL to transform data in the batch store; it has a version control system for feature definition, and users can publish new versions using CLI tools; users (or schedulers) can execute commands to load features from offline store to online store; at the training stage, users use Feast Python SDK to retrieve datasets for training and evaluation; Feast also supports point-in-time feature retrieval to get correct historical features; model serving services use Feast SDK to get online features.

Feast natively supports three kinds of offline stores: File, BigQuery, and Redshift. For the simplest use case, you can store offline features in a local file system as Parquet files. For the online stores, Feast supports SQLite, Redis, Datastore (GCP), and DynamoDB (AWS).

Feast abstract a concept called *Provider*. A provider is an implementation of a feature store using specific feature store components (e.g., offline store, online store) targeting a specific environment (e.g., GCP stack). For example, Feast natively supports three kinds of providers: Local, GCP, and AWS. For the Local provider, the offline store uses File by default and online store uses SQLite. For the AWS provider, offline store uses Redshift and online store uses DynamoDB. This gives users flexibility to deploy their feature stores on different cloud services.

Feast starts as an open-source project with a community that is close to Kubeflow. Recently, the founder and core contributor of Feast joined Tecton.ai, a startup company focusing on commercial feature store products.

Databricks Feature Store

As one of the most famous cloud service providers that focus on data and machine learning, Databricks is also building their own feature store.

Databricks Feature Store is a centralized repository of features. It enables feature sharing and discovery across your organization and also ensures that the same feature computation code is used for model training and inference. It is not open-source and is only available on Databricks Platform. It is now in Public Preview.

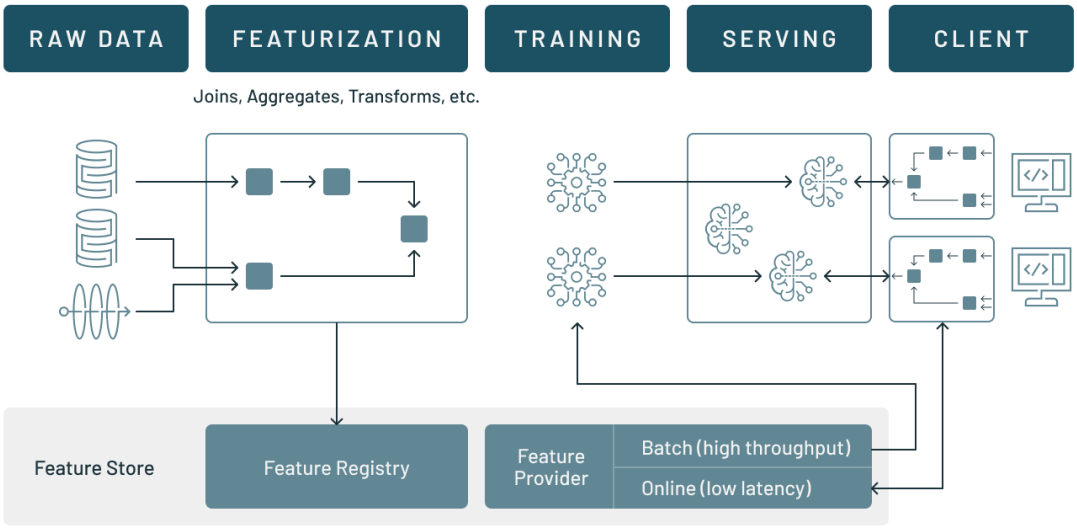

Figure 3 Databricks Feature Store Architecture

Since Databricks is the company behind many widely adopted data products such as Spark, Delta Lake, and Koalas. It is natural to see that Databricks Feature Store is fully integrated with other Databricks components. Offline features are stored as Delta tables and online store supports Amazon Aurora and Amazon RDS MySQL.

Much like other products of Databricks, Databricks Feature Store heavily relies on Notebook to interact with. Thus, its API is Python-centric. Typical machine learning workflow using

Databricks Feature Store can only involves writing Python in Notebooks. It also provides a simple GUI allowing users to search for features, identify data sources, and control access to the feature. As a product targeting business users, it also supports access control to features.

SageMaker Feature Store

As an important product for AWS to capture MLOps market, SageMaker also announced their fully managed feature store service. Amazon SageMaker Feature Store is a purpose-built repository where users can store and access features so it's much easier to name, organize, and reuse them across teams. SageMaker Feature Store provides a unified store for features during training and real-time inference without the need to write additional code or create manual processes to keep features consistent.

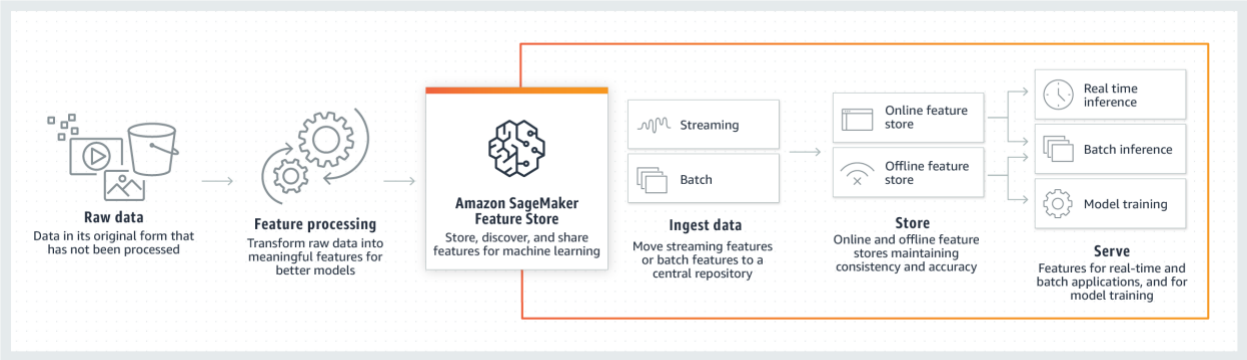

Figure 4 SageMaker Feature Store Architecture

SageMaker Feature Store allows users to ingest features from many sources using Kafka or Kinesis. The offline data is stored in S3 bucket as Parquet files. It also provides a rather sophisticated UI for feature discovery and management. SageMaker Feature Store helps with feature sharing across teams by storing feature definitions in a single repository so that it is clear how each feature is defined. It also integrates well with Amazon SageMaker Pipelines.

SageMaker Feature Store is not open-source and it is only available on AWS. It is also not clear what database is used for online store.

MOST IMPORTANT FEATURES

After analyzing these products, several patterns emerge.

Almost every feature store has dual-store architecture: a K-V store for online store and a data warehouse or data lake as the offline store. It is chosen based on the nature of data access pattern. For the model training, features are queried in a large volume (millions or billions of rows per query) with possibly exploratory data analysis. While for the online inferencing, features are retrieved in a small batch (hundreds of rows) in a real-time, low latency environment. It is similar to the two kinds of data access pattern: OLTP and OLAP. Because of the different nature of two access patterns, it is more suitable to use different databases for each one.

One of the most centric concern of feature store is feature consistency. Because of the dual-store architecture, we have to establish some kind of sync mechanisms to ensure consistency between two stores. Feature consistency is important for machine learning application.

Consistent feature engineering library is proposed. There are many ways to ensure consistency. One way is to apply the same feature engineering logic when ingesting raw data on both online and offline stores. It is not an easy thing to do because data source of online and offline features is radically different. For example, offline features are most often from layers analytical tables that data analysts built using complex SQL; while online features are often from API calls to services, database query, or request context data. Some feature store proposed a feature engineering toolkit library (using Python mostly) to unify two ingestion pipelines. These libraries often have some abstractions (such as Feature Table, Feature Entity, Feature Group) that can incorporate different kinds of data source and format.

One of the most important functions that offline store provides is Time Travelling. The reason is that when producing dataset for model training, we have to make sure the feature we use is the same as the feature at the time when the event happened. Otherwise, we may mistakenly incorporate label information into the feature causing the model to learn incorrect latent relations. For example, if we are predicting whether the user is going to buy product, we have to use the feature of the user before he/she made the purchase. Otherwise, the feature may already contain information indicating that the user likes the products (because he/she purchased it) and we are using that to predict the purchase. In order to prevent this kind of inconsistency, we have to be able to let the user retrieve features of certain time (as in time travel). It is also known as point-in-time join, as joining labels with features with matching timestamps. To accomplish this, we have to have some versioning mechanisms in offline store because features are constantly changing and we need to save all historical features. Many data warehouse or data lake products already provides such ability. Delta Lake is one of them.

Features should be sharable among teams. One of the reasons that feature store is proposed is that different teams in a large company want to share features. Prior to feature store, they have to build their feature processing stack and write feature engineering logic independently. Many labors are wasted in reinventing the wheels. Feature store provides a centralized feature managing and discovering registry, allowing users from different teams to publish and subscribe features.

Python SDK is the central API. Due to the popularity of Python in ML community, many feature stores choose to put Python SDK in the first place. If you are training model using TensorFlow or PyTorch, it is easier to embed your dataset retrieval script into your training script. Although many existing data analytics pipelines are built using SQL, SQL code is hard to directly translate to transformation logic in online ingestion which requires real-time processing.

SYSTEM DESIGN

Overview

For the scope of the course project, we are going to build a toy feature store called Rice Simple Feature Store (`rice-sfs`). It contains the following components:

- Offline Store: Delta Lake, features stored in Parquet
- Online Store: DynamoDB
- Ingestion Engine: Spark for batch ingestion and Spark Streaming for streaming ingestion
- Feature Registry: Spring MVC service with REST API connected to PostgreSQL
- Feature Serving: Spring WebFlux service serving features stored in Online Store

Architecture

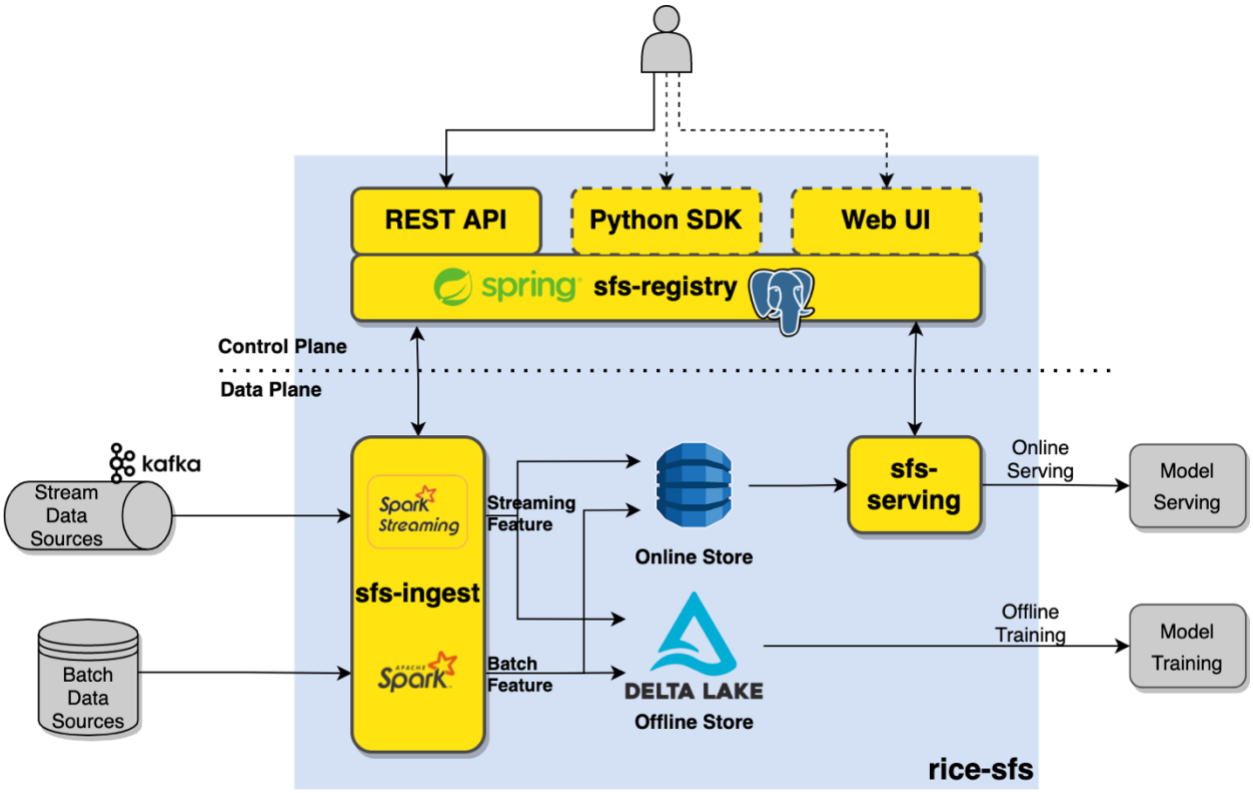

Figure 5 Rice Simple Feature Store Architecture

The whole system is divided into two parts: control plane and data plane.

In the control plane, we have the core component, feature registry(`sfs-registry`), the metadata management center of our feature store. Its responsibility is to manage all the feature schemas, versions, and other metadata. It uses PostgreSQL as the persistent database. It provides a RESTful API for ingestion engine and other services to query feature schema and other metadata. It also serves as the backend for future possible Web UI or Python SDKs.

In the data plane, from left to right, the features are ingested into the system and consumed by the down-stream systems.

First, we have an ingestion module(`sfs-ingest`) to ingest features from various data sources including batch sources and streaming sources. It uses Spark and Spark Streaming as the compute engine.

The ingested feature would be store in two data stores: Offline Store and Online Store. We choose Delta Lake for offline store because it provides convenient way to versioning the feature and it has great compatibility with Spark, our feature ingestion engine. We choose DynamoDB as the online feature store because it provides a fast k-v store with persistency and high throughput. Another reason is that we want to try some trending cloud services.

To access historical features from offline store, user can directly access the Delta Lake and query the data. For the online feature, we have a service(`sfs-serving`) to serve features from the Online Store. It uses WebFlux to maximize concurrency capacity.

Concepts

We adopt a similar abstraction as the Feast to organize features in the system.

- `Entity`, represents a domain entity in the users' business, such as "User" and "Movie".
- `Feature`, represents a single feature. It has a name and a type. For example, "user_age" is a Feature of `INT` type and "user_gender" is a Feature of `STRING` type.
- `FeatureTable`, is a collection of Features always updated together. Features in a FeatureTable are physically stored together. Each FeatureTable has exactly one Entity used as the joining key. For example, "rating", "length", and "watched_times_a_week" are three Features of Entity "Movie" in a FeatureTable.
- `FeatureView` is a logical collection of Features in a FeatureTable that are often used together in a model.

COMPONENTS

Next, we would dive deep into the implementation of the system. In this section, we would take a detailed look of the major components of the system to see how they are implemented.

Feature Registry

The Feature Registry is the core component in the system. It runs in the control plane. It serves as the central controller of the whole system, like a brain. All other components in the system talk to the feature registry to retrieve metadata.

The most important function of the feature registry is to manage the metadata, the concepts we mentioned above: Entity, Feature, FeatureTable, and FeatureTableView.

Feature Registry provides a rich set of RESTful APIs for the end users and other components in the system to interact with:

```
GET     /api/v1/entities
POST    /api/v1/entities
GET     /api/v1/entity/{{name}}
PUT     /api/v1/entity/{{name}}
GET     /api/v1/feature/{{name}}
PUT     /api/v1/feature/{{name}}
POST    /api/v1/featureTable/{{featureTableName}}/feature/{{featureName}}
DELETE  /api/v1/featureTable/{{featureTableName}}/feature/{{featureName}}
GET     /api/v1/featureTable/{{name}}
PUT     /api/v1/featureTable/{{name}}
GET     /api/v1/featureTableView/{{name}}
DELETE  /api/v1/featureTableView/{{name}}
GET     /api/v1/featureTableViews
POST    /api/v1/featureTableViews
GET     /api/v1/featureTables
POST    /api/v1/featureTables
GET     /api/v1/features
POST    /api/v1/features
```

With these APIs, user can easily manage the metadata by creating, updating, retrieving, and deleting Entity, Feature, FeatureTable, and FeatureTableView.

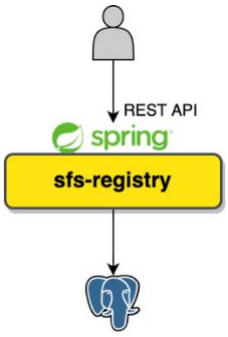

Figure 6 Feature Registry

Feature Registry stores the metadata in a PostgreSQL database. The schemas of the tables are:

```
CREATE TABLE IF NOT EXISTS entity
(
    name        VARCHAR(256) PRIMARY KEY,
    description TEXT
);
CREATE TABLE IF NOT EXISTS feature
(
    name                 VARCHAR(256) PRIMARY KEY,
    description          TEXT,
    value_type           VARCHAR(20),
    delta_table_col_name  VARCHAR(256),
    dynamo_table_col_name VARCHAR(256)
);
CREATE TABLE IF NOT EXISTS feature_table
(
    name              VARCHAR(256) PRIMARY KEY,
    description       TEXT,
    entity            VARCHAR(256),
    features          VARCHAR(256)[],
    delta_table_path  VARCHAR(256),
    dynamo_table_name VARCHAR(256)
);
CREATE TABLE IF NOT EXISTS feature_table_view
(
    name               VARCHAR(256) PRIMARY KEY,
    feature_table_name VARCHAR(256),
    feature_names      VARCHAR(256)[]
);
```

Note that for each Feature, we have to specify its value type. This allows us to do the schema verification when ingesting the feature. We also have to specify column names in both delta table and dynamo table, delta table path, and dynamo table name so that other components know where to store/load the feature.

When creating a new FeatureTable, Feature Registry would also try to create a new dynamo table in the connected DynamoDB so that ingestion module would not need to create table before ingesting into the online store. Delta table is created automatically when writing data into, so we do not need to create it manually.

Feature Registry uses Spring Data to interact with PostgreSQL server. Spring Data provides a high-level abstraction of relational database tables so that we can focus on the business. Besides, with this abstraction, we can easily port the system out to another RDMS such as MySQL without changing much of the code.

Feature Registry users Amazon DynamoDB Java SDK to connect to DynamoDB server. Specifically, we are using Document Interface because it provides a sweet spot between flexibility and convenience.

Feature Ingestion

The Feature Ingestion module helps to ingest data into the system. It serves as the gateway of the data plane. All the features have to go through the ingestion pipeline in order to get into the system.

Essentially, the feature ingestion module is just a set of Spark jobs written in Scala with pre-defined structure that can load features from some data source and write to the two data stores (offline and online).

When submitting an ingestion job, users have to provide some information about the data source. For example, when ingesting from a batch data source, users would have to provide the following configs:

```
options: Map[String, String],
format: String,
path: String,
cols: Seq[String], // columns in the original file
featureTable: String, // name of the feature table
entityCol: String, // entity col name
featureColMap: Map[String, String], // feature name -> col name
```

The Spark Job would use this config to run the ingestion workflow.

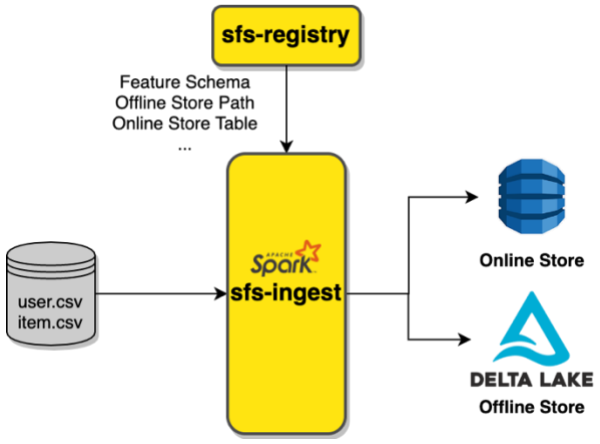

Figure 7 Feature Ingestion

First, it would send a request to the Feature Registry asking the information of the feature table it is working on. The response would contain all the metadata of the feature table, including the entity, all the features, delta table path, and dynamo table name.

Then, it would try to load the feature from the data source into a DataFrame with the path and format provided above. Users would also have to provide the name of each column in the original data source if it is not encoded in.

After that, it would validate the schema by explicitly casting each column into the desired data type. The desired data type is computed via a mapping from the ValueType users defined when registering the features to the Spark DataType:

```
def valueTypeToSparkType(v: ValueType): DataType = {
  v match {
    case BYTES => BinaryType
    case STRING => StringType
    case INT => IntegerType
    case LONG => LongType
    case FLOAT => FloatType
    case DOUBLE => DoubleType
    case BOOL => BooleanType
    case UNIX_TIMESTAMP => TimestampType
    case BYTES_LIST => createArrayType(BinaryType)
    case STRING_LIST => createArrayType(StringType)
    case INT_LIST => createArrayType(IntegerType)
    case LONG_LIST => createArrayType(LongType)
    case FLOAT_LIST => createArrayType(FloatType)
    case DOUBLE_LIST => createArrayType(DoubleType)
    case BOOL_LIST => createArrayType(BooleanType)
    case UNIX_TIMESTAMP_LIST => createArrayType(TimestampType)
    case _ => NullType
  }
}
```

Finally, it would rename the columns into the specified column name in the feature metadata for delta table and dynamo table and write the DataFrame to the two data store.

We use the official jar provided by Delta Lake `io.delta:delta-core_2.12:1.1.0` to write into delta lake. We use append mode when writing the data. We also added a special column "`__updated_at__`" to record the timestamp of the ingestion.

For the ingestion of the online store, we use a third-party Spark-DynamoDB connector `com.audienceproject:spark-dynamodb_2.12:1.1.2` to batch write DataFrame into DynamoDB.

Feature Serving

After ingesting the feature, we need to provide a convenient way to allow users to retrieve features from the system.

For the training stage, users can directly access the offline store via Spark Jobs or other batch query engines. Users can directly query Feature Registry to see where is the feature store and what features can they use.

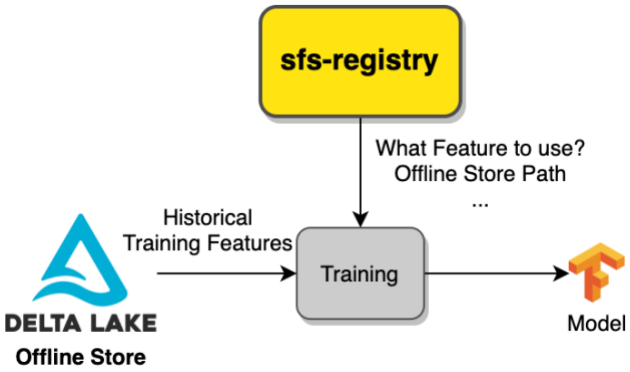

Figure 8 Training using SFS

When it comes to the serving stage, we use a serving service to provide online features. The Feature Serving module is a Spring WebFlux application that serves as the serving endpoint of the system.

Users would have to first go to the Feature Registry to register a FeatureTableView which represent a logical view of a FeatureTable. In the FeatureTableView, users specify what feature table they are using and a subset of features they are interested in. Then, users can directly query the Feature Serving service with the name of the feature table view and id of the entity:

```
GET /api/v1/batchGetFeature
GET /api/v1/getFeature
```

When getting the request, serving service would first ask the Feature Registry for the detailed metadata of the feature table along with the specified view. It would query the online store, DynamoDB, with this information. We use Projection Expressions to avoid retrieving the entire document when users are only interested in a subset of the feature.

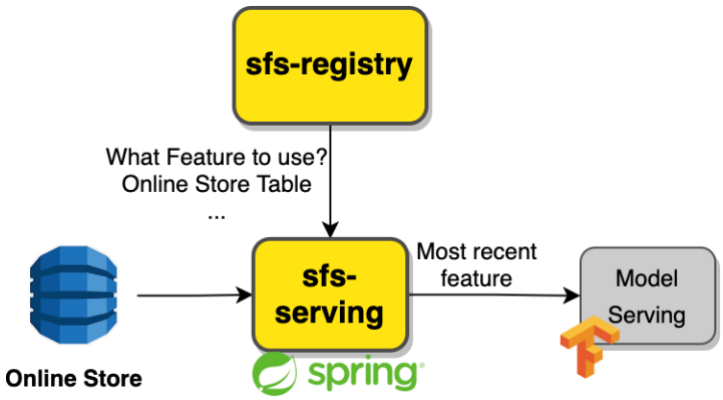

Figure 9 Feature Serving

The result features can then be used in a mode inferencing service.

DESIGN DECISIONS

In this section, let's take a look at some of the most interesting design decisions we made and discuss how they contribute to the overall goal of the feature store system.

Why uses Java?

Machine Learning field is dominated by Python. You may wonder why we choose to use Java to implement a system that is mainly used by ML community.

The main reason is that, although serving the ML workflow, feature store is mainly about data processing and management rather than the actual model training and serving. Feature store systems have a close relation with many "Big Data Tools". As we all know, many big data tools are written in Java (or at least in a JVM language such as Scala). We choose to implement the system in Java because we want to have a seamless interaction with other big data tools such as Spark, Flink, and Kafka. For example, the ingestion module in our system is mainly just Spark Jobs written in Scala. We can share the models and some of the HTTP clients with other components.

How to ensure consistency?

One of the most important features of feature stores is to ensure consistency between offline training feature and online serving feature. How does our system ensure feature consistency?

Basically, we do it by restricting to a single source of data. Features in offline store and online store are all come from a single source, ingestion module. When ingesting data, we restrict that if a set of features is about to be ingested into the offline store, it must also be synced to the online store and vice versa. By controlling the source of the feature, we can ensure that features in the two data stores are always consistent.

Furthermore, we would also do the schema validation when ingesting the data. This also helps with ensuring the consistency of the feature in a sense that the type of the feature is visible to the end users and consistent.

How to time travel?

When querying the offline store to retrieve features for model training, users tend to want to be able to get features from a specific time snapshot, just like time travelling to that timestamp. This is also one of the most essential features of feature stores. How does our system do that?

This would come down to the reason of using Delta Lake as the offline store. It turns out that Delta Lake itself record all the historical data and can easily go back to a certain version of the data. With Delta Lake, we automatically provide the ability for time travelling in the offline store.

DEMO

We demonstrate how to use the system by walk through an example of movie recommendation application. Please go to the "example/" directory of the project repository and check out he Jupyter Notebook. You can also see a PDF version of the demo report.

LIMITATIONS

Besides the seemingly full functionality set that Rice Simple Feature Store provides, the system is still just a toy with many limitations.

First, it is designed to be tightly coupled with specific online store and offline store. We hard-coded it into the system and even the scheme of the metadata. We also heavily rely on Delta Lake providing the functions of time travelling. From this prospective, we may say that the system is not very extensible. In a real-world system, especially with multiple users(companies), the choice of data stores should be more flexible because different customers may have various dependency limitations.

Second, it does not support any internal feature transformation functions. If you want to do some pre-processing to the features, you have to do it outside of the feature store system before ingestion. This decision is for simplifying the design of keeping feature consistency because when you don't have to do the transformation, you don't have to make sure the transformation logic is consistent. Besides, providing a feature transformation function means we have to provide some kind of programmability to the user, which probably means we have to have some kind of DSL. This work is far beyond the scope of this project. It is worth mentioning that real-world systems tend to provide this function because it is more convenient for the customers.

Last, the system is designed without any consideration of the security in mind, which means the system has many potential security issues. For example, it would be better if we have an access control system for the users when operating on the feature registry. There are also many corner cases that are overlooked, such as how to deprecate a feature or what would happen if some of the features get corrupted. In real-world systems, more concerns are given on security but there are definitely some security issues there in some of the most popular feature stores.

FINAL THOUGHTS

By doing the analysis of the feature store products on the market, I gained a big picture of the domain; by actually design and implement a simple feature store, I get to have a deeper understanding of it. Although the system is far from production-ready, it already connects to many domains of cloud computing.

Building a feature store is like building a data analysis platform. In order to make it on the market and sell it to many customers, you have a tons of integration work to do. There are too many systems you need to connect to in order to cover the requirements of most potential customers. The core of the feature store is actually not that complicated. The problem is that the boundary of the system is blurred.