tecton

The Data Platform for Machine Learning

Summary

This document discusses common components of basic feature stores and their associated management considerations/overhead. It then goes on to discuss advanced capabilities that are commonly required from production-ready feature stores.

Basic Feature Store

A basic feature store materializes features on a batch schedule. It is also able to serve feature values to data scientists for training and to operational systems for low-latency serving.

Capabilities

Such a Feature Store typically offers the ability to:

Define
- Feature Transformation pipelines

Orchestrate
- The execution of incremental Feature Pipelines on external data processors (like Spark)

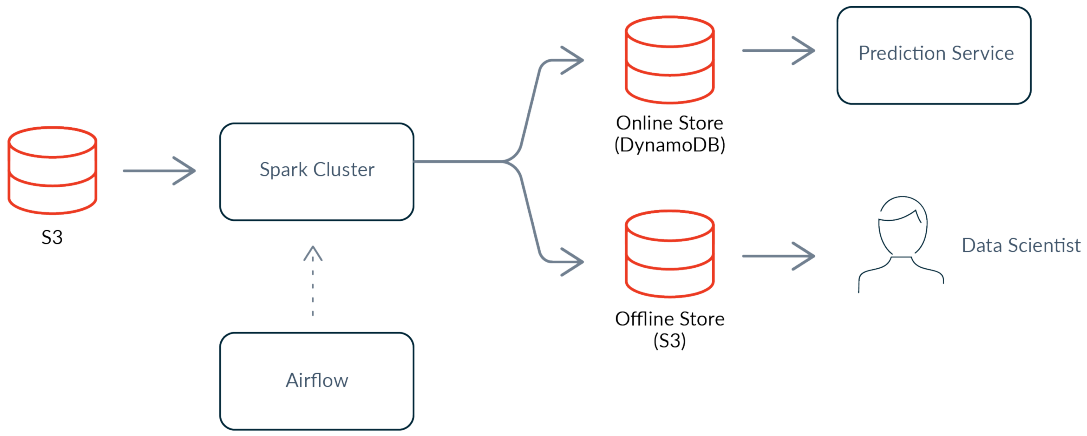
Store
- Materialized feature values for offline and online serving

Serve
- Most recent features at low-latency **online** and in **multiple regions**
- Historical feature values **offline**

Monitor
- Feature pipeline health—Are they running and finishing? Are they crashing or stalling?
- Feature pipeline output—Are features getting stale?
- Feature serving system—Is the data store and the serving system properly provisioned? Are latencies and availability healthy?

Operate
- Backfill the store when data issues arise
- Scale data pipelines, data stores and serving systems with increased load

Sample Implementation



At a high level:

- Every feature is defined as a separate Airflow DAG
- Those DAGs run on a specific schedule (eg hourly) and kick off Spark jobs on a Spark cluster
- Those Spark jobs process the raw data and write feature values to a Dynamo table (for online serving) and S3 (for offline serving)
- The prediction service integrates with DynamoDB and knows in which table it finds specific features and reads values directly
- The data scientist knows in which S3 location (or Hive table) to look for historical feature values

Some Things to Keep in Mind

Define Features

- The data scientist needs a way to create those Airflow DAGs which describe the feature transformations. So they need to not only be concerned with the feature transformations, but also the Airflow framework
- The data scientist needs a way to push those DAGs to production
- It should be possible for the Data Scientist to experiment with a feature transformation before committing the transformation to production in the form of an Airflow DAG

Orchestrate Feature Pipelines

- A Spark cluster needs to be running somewhere so that Spark jobs can be executed (or be spun up by Airflow DAGs on-demand)
- Those Spark clusters need to support the provisioning needs of the Feature Transformation DAGs
- The Airflow DAGs need to be able to discover and communicate with the Spark Cluster to submit the Spark job to them
- The Spark Cluster must be able to read from the raw data source (eg auth and discovery needs to be handled)
- Spark Job errors need to be noticed, retried and broadcasted to an alerting system
- The output of the Spark Jobs should be monitored to ensure that upstream data is coming in
- There should be a way to kick off backfill runs to overwrite feature values in the feature store

Store Features

- DynamoDB scales capacity elastically—retries are important to implement to ensure feature values don't get lost (e.g. writes will fail while Dynamo is scaling)
- Ensure that your stored feature values are associated with an event-time derived timestamp. Otherwise it's easy to have historical backfills, or retried materialization jobs, suddenly overwrite the most recently served feature values

Serve Features

- The Data Scientist needs to know where to retrieve historical feature values to generate training datasets. For example, they might register features in Hive tables
- If multi-region serving is required, Dynamo's multi-region replication can be a good starting point
- Keep in mind that online serving stores have their own limitations to work around (e.g. DynamoDB's hot partitions don't support >3000 reads per second)

Monitor

- Frameworks like Great Expectations can be used to ensure some base level health of the data pipelines

Enterprise Feature Store Capabilities

An advanced feature store builds on the minimum feature store and offers the following capabilities on top:

Transformations

Real-Time Transformations: The ability to execute real-time transforms consistently online and offline

- It is important to run the same transformation in your online serving system when you request feature values as when you generate training data.
- To achieve consistency, you need to make sure that the exact same transformation code gets executed in your serving system.

Streaming Transformations: Features that have freshness requirements of <1h should really run off of streaming systems

- You need a way to backfill those features from batch sources and then steady-state materialize features from streaming pipelines. Finding the right hand-off point is tricky
- Further, you need to find a way to kick off streaming jobs and keep those running. A periodically running Airflow DAG may do the job here that restarts Streaming jobs

Reduce Processing and Storage Costs

- For example, if you use a lot of time window aggregations as features, you can dramatically reduce processing and storage costs by storing aggregations in "tiles". E.g. imagine you have a 1day, 5day and 10day trailing purchase-count feature. The 5day and 10day trailing purchase-count feature can be calculated by summing up multiple 1day feature values.
- This implementation allows you to add new features of different time window spans essentially for free—no back fills are necessary and you can use them in production immediately
- Further, if you ever need streaming time window aggregations of time windows that are too large to fit in memory, you can't get by without implementing some form of tiling yourself (the streaming job will eventually run out of memory)

Ability to seamlessly leverage Spot Instances for Processing

- A system that uses Spot instances for data processing needs to be able to differentiate between true data processing errors and ephemeral ones that are caused by preempted Spot Instances. Otherwise, you either pay the extra cost for reserved/on-demand instances to get greater stability, or you deal with lots of noisy alerts.

Metadata Management

Versioning of Features

- You need to make sure that changes to feature definitions don't write into the same feature store tables. Otherwise you'll quickly end up training models by accident on different versions of features. This can quickly lead to poor model performance, because online you'll typically only serve the latest feature version
- Ideally, you'll also find a way to tie the version of a model that's running in production back to an exact feature definition so that you can always reproduce a model

Central Catalog and Sharing

- If you have more than two data scientists, there should be an easy way for data scientists to leverage and discover each other's work
- Further, there should be an easy way for data scientists to fork and reuse each other's features
- At smaller scale, a shared git repo may be good enough. At larger scale, tools like LinkedIn's DataHub or Lyft's Amundsen can be used, although they are general to data discovery and not specific to machine learning features

Lineage Tracking

- Enterprise Feature Stores should be able to track the full lineage of features. What upstream data sources and transformations (and their versions) does the transformed data come from?
- That's often important for full reproducibility of models, compliance and even understanding a company's dependency graph

Operations

Data Quality and Data Drift Monitoring

- Models don't break—it's the data that breaks. An Enterprise Feature Store should monitor the raw data that's batch and stream processed by your Spark jobs
- Feature values should be monitored as well. Are they starting to drift? Do the statistics of the features look similar to the statistics of the features you trained your model on?

Experimental and Staging Environments

- Ideally you can provide your data scientists with a safe environment to develop and experiment with new features without risking to break production
- In an ideal world, data scientists can spin up their own new environment at no additional cost (Tecton calls this Workspaces)
- Early on, you may be able to get by with a separate staging instance of your Feature Store

Cost Visibility

- Enterprise Feature Stores should give you a detailed breakdown of write-materialization cost of features (e.g. cost of processing jobs)

- The read cost of features (e.g. cost of reading data from Dynamo)
- The storage cost of features (e.g. Dynamo and S3 storage cost)
- Further, it should be easy to overlay those costs with usage information. Are you spending money on a feature nobody uses? What models benefit from a feature? Is the model's bottom line impact worth the cost of the feature?

Data Science Experience

Feature Calculation from Raw Data

- Not every feature deserves to be continuously materialized (e.g. features that are only infrequently requested for training and features that don't require online serving). Some are good enough to materialize ad hoc. It should be easy for data scientists to still register, discover and use those features.

Time Travel

- To avoid data leakage, data scientists need to know what the world looked like at a certain point of time in the past. It's important to differentiate between record time and event time. If a data scientist has a specific point of time in the past, there needs to be a way to retrieve the most recent feature value as of that point of time in the past.
- Something like Hudi's or Delta Lake's time travel allows you to travel to one point of time in the past. But those systems don't work for continuous time travel that you need for time series models. You can look into Flint to support fuzzy time series time joins for problems of this kind. This capability needs to work at scale, because models trained on time series need to know what the world looked like at a large number of different points of time in the past.

Bundling multiple features for models

- A good feature store should be able to serve not just individual features. Models typically require a set of multiple features. There should be a way to bundle multiple features into a group of features that map to a specific model. Tecton calls this a FeatureService. These bundles should be versioned as well.

Integration with the broader data science ecosystem

- ML workflows are best solved by using a combination of best-in-class tools, frameworks and platforms. A Feature store should natively fit in with a cloud native data science stack (Sagemaker, EMR, MLFlow, Azure ML Workbench, Seldon Core, etc.)

Automation

DevOps

- Ideally, the best DevOps practices are used to manage your feature pipelines. CI/CD pipelines should test your feature pipeline code that you check into git, roll out changes to production and continuously monitor the health of those pipelines.
- Integration with a company's broader data architecture
- A feature store runs transformation pipelines. Likely, a company has more than just ML feature data pipelines. A good feature store should make it easy to integrate with the existing data stack of a company, supporting capabilities such as:
    - Triggered feature materialization via an API
    - Custom metadata management

Compliance & Governance

ACLs

- Not every data scientist will have access to the same data

Audit Logs

- For compliance reasons, it's often required to create and persist audit logs of feature access by various users

SSO

- If you use a central SSO system like Okta, you can give data scientists access to the Feature Store and to certain features based on your single sign-on system
- Many B2B companies require SSO to comply with the security requirements of their own customers

GDPR Compliance

- For GDPR, there should be an easy way to delete individual records from the online and offline feature store
- Deleting a key from Dynamo is relatively easy
- Deleting a key from the offline feature store is trickier. You can use a system like Hudi or Delta Lake for incremental deletion. You need to know if tomb-stoning the data is enough or if you require the data to be permanently deleted

Support

Multi-Cloud Support

- To help mitigate vendor lock in, advanced feature stores should make it possible to seamlessly support not just one but all the major cloud providers (AWS, GCP, Azure)

SLA and Support

- This is of course less of a capability but instead a guarantee. But at some scale, an advanced feature store should come with a supporting team that provides latency and uptime guarantees for your business critical applications