Feature Store Implementation for Real Time Recommender Systems

Geonwoo Cho

Hyperconnect

Contents

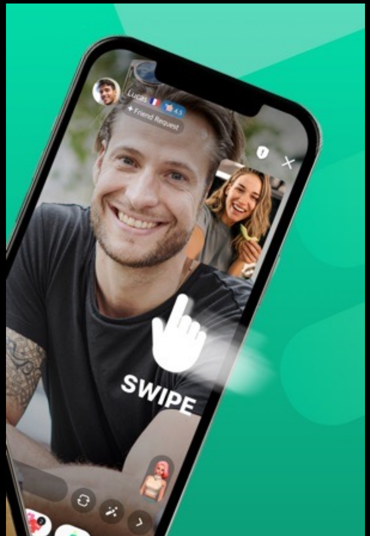
1. Recommender Systems of Hyperconnect
2. Preliminaries
3. Why was the Feature Store Introduced?
4. Feature Store of Hyperconnect
5. Case Studies & Adoption Impacts

1. Recommender Systems of Hyperconnect

1.1 Recommender Systems of Hyperconnect

Azar

A social service connecting people worldwide with just one swipe.

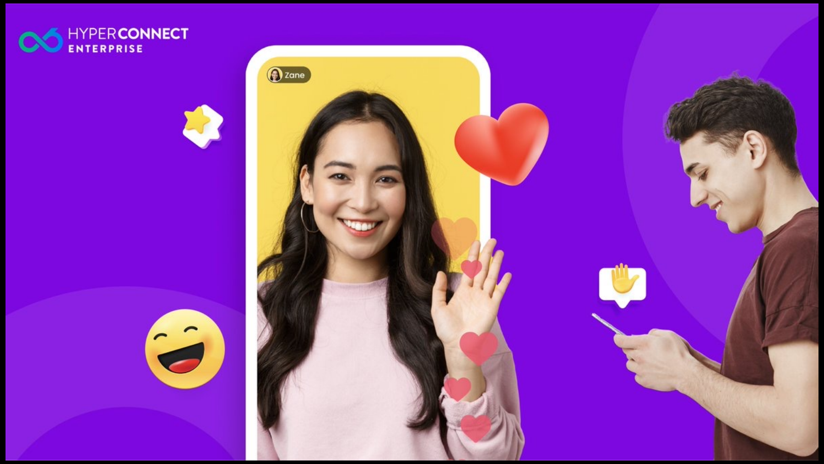


Hakuna

An interactive social live streaming platform where anyone can freely participate in broadcasting

Hyperconnect Enterprise

B2B solutions leveraging HyperConnect's accumulated video technology."



1.1 Recommender Systems of Hyperconnect

Azar

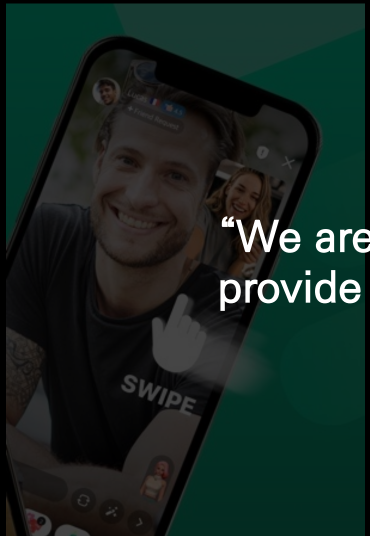
- 1:1 matching
- Lounge Card recommendation
- Live broadcast recommendation

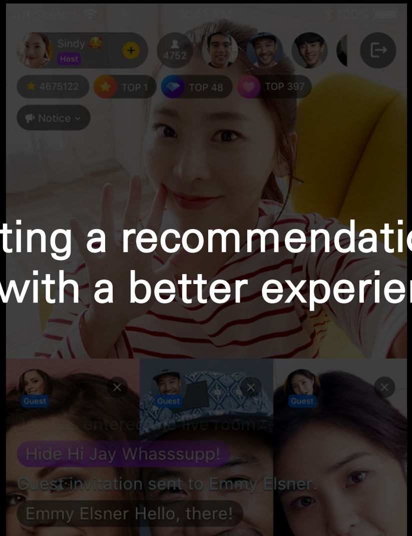
Hakuna

- Live broadcast recommendation
- popular streamers recommendation

Hyperconnect Enterprise

- Live broadcast recommendation (B2B solution)





"We are operating a recommendation system across various features to provide users with a better experience while connecting people."

1.2 Differences Compared to Other Recommender Systems

Item = User

A specialized domain that recommends users, as opposed to typical recommendation systems that recommend static items like products or content

Real Time Recommender Systems

- Hyperconnect's recommendation system is primarily constrained to recommending online users.
- "In user-to-user recommendation systems, if both users are new (cold), the system may show very low recommendation performance unless real-time data (such as real-time action logs, context, etc.) is used."
- Hence, a recommendation system that considers real-time aspects is essential.

2. Preliminaries

2.1 Recommender Systems and features

Dataset

<User features>					<Item (peer user) features>					<Target>
id	gender	country	age	avg_chat_sec	id	gender	country	age	avg_chat_sec	chat duration
1001	MALE	KR	20	12	2001	FEMALE	KR	21	3	142
1001	MALE	KR	20	12	2002	FEMALE	JP	23	3	5
1002	MALE	JP	26	142	2002	FEMALE	JP	25	71	35
1002	MALE	JP	26	142	2003	FEMALE	CA	21	11	51
1003	FEMALE	US	22	48	2001	FEMALE	KR	21	3	11
1003	FEMALE	US	22	48	2002	FEMALE	JP	23	71	121
1003	FEMALE	US	22	48	2003	FEMALE	CA	25	11	26

Training to predict the target (chat duration)

ML/AI model

Unknown Input (Online Environment)

1001	MALE	KR	20	12	2003	FEMALE	CA	25	11	???

Predicting how long the conversation will last when user 1001 meets user 2003 (inference).

2.1 Recommender Systems and features

Dataset

<User features>					<Item (peer user) features>					<Target>
id	gender	country	age	avg_chat_sec	id	gender	country	age	avg_chat_sec	chat duration
1001	MALE	KR	20	12	2001	FEMALE	KR	21	3	142
1001	MALE	KR	20	12	2002	FEMALE	JP	23	3	5
1002	MALE	JP	26	142	2002	FEMALE	JP	25	71	35
1002	MALE	JP	26	142	[illegible]	FEMALE	CA	21	11	[illegible]
1003	FEMALE	US	22	48	2001	FEMALE	KR	21	3	11
1003	FEMALE	US	22	48	2002	FEMALE	JP	23	71	121
1003	FEMALE	US	22	48	2003	FEMALE	CA	25	11	26

For online model inference, it's necessary to load features corresponding to users/items online.

Unknown Input (Online Environment)

1001	MALE	KR	20	12	2003	FEMALE	CA	25	11	???

Training to predict the target (chat duration)

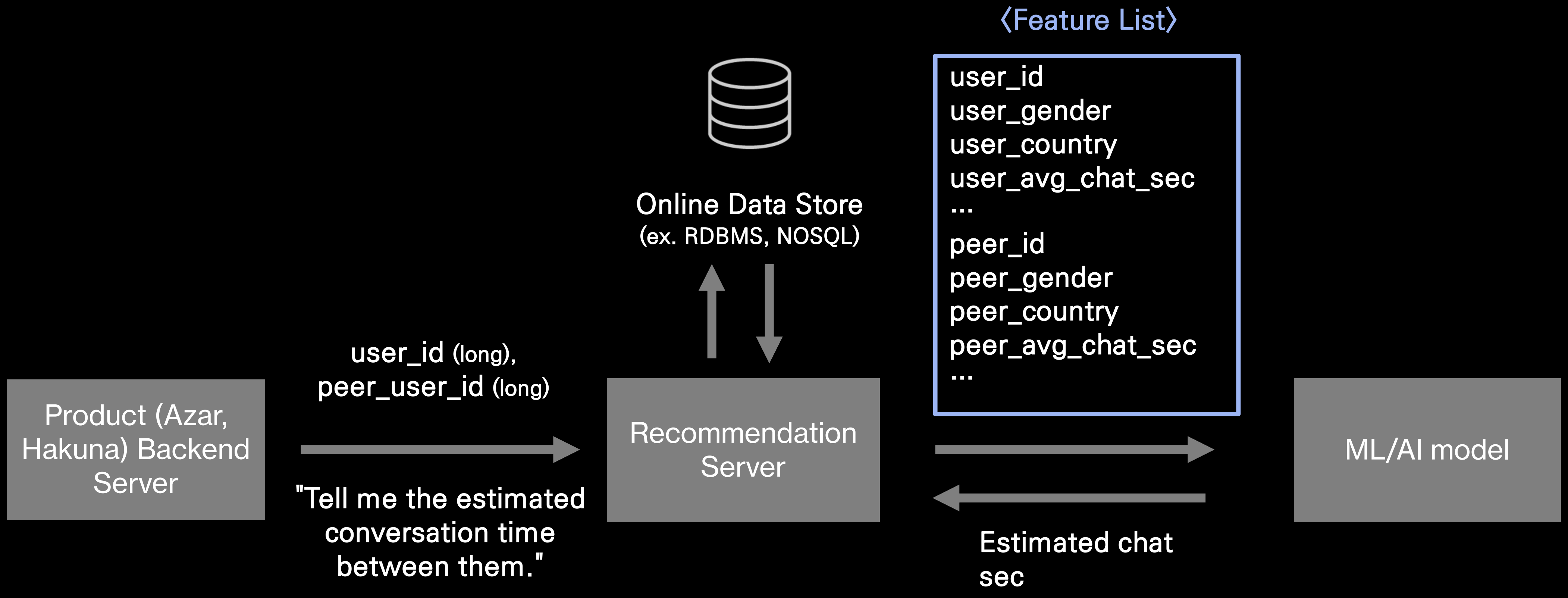
ML/AI model

Predicting how long the conversation will last when user 1001 meets user 2003 (inference).

2.1 Recommender Systems and features

- For online recommendations, a data storage system capable of loading feature data is required in the online environment.

- Challenge: The same features must be stored in both offline (BigQuery) and online (RDBMS, NoSQL) storage systems.



2.2 Data Storage Technologies for Machine Learning Applications

– In recommendation systems, online data needs to be loaded for model inference in serving logic
Ex) gender, country code, birthday, registration date, average purchase amount, etc

– Implementing a storage solution that satisfies the characteristics of both training data and serving data can be challenging.

	Training data	Serving data
Read pattern	Accessing multiple records based on timestamps	Accessing specific data based on key.
Query Frequency	Occasional, Periodic	Frequently
Latency	-	Fast
Throughput	High	-

2.2 Data Storage Technologies for Machine Learning Applications

Training data query

```
SELECT gender, count(*)
FROM user_profile
GROUP BY gender
```

- Accessing single/few columns.
- Accessing multiple records where latency is not crucial, hence the importance of caching is low.

Serving data query

```
SELECT *
FROM user_profile
WHERE user_id=1234
```

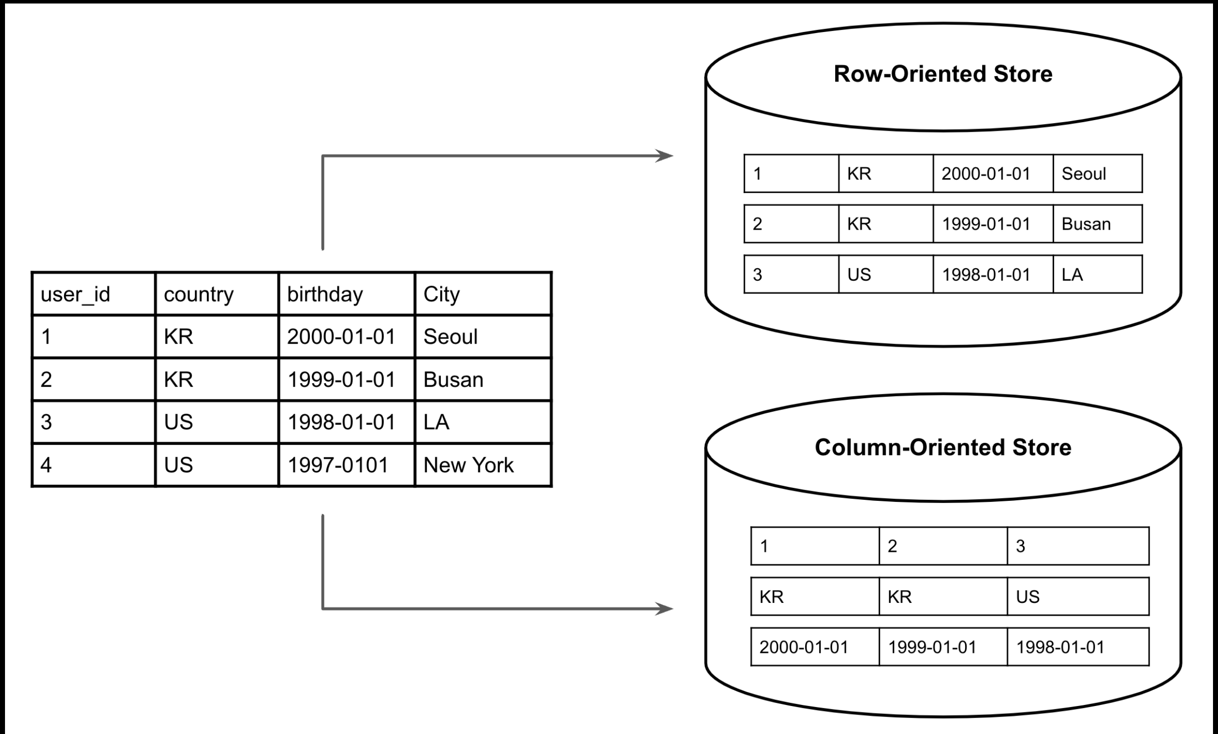
- Accessing multiple columns.
- Utilizing cache to reduce latency since records are divided into frequently accessed (hot) and infrequently accessed. (cold)

2.3 Database for Training data: Bigquery

- Column-oriented database

- Advantages in data access patterns at the column level, but inefficient for searches on individual records.

- Column data often has higher data redundancy than row data, leading to better compression efficiency.



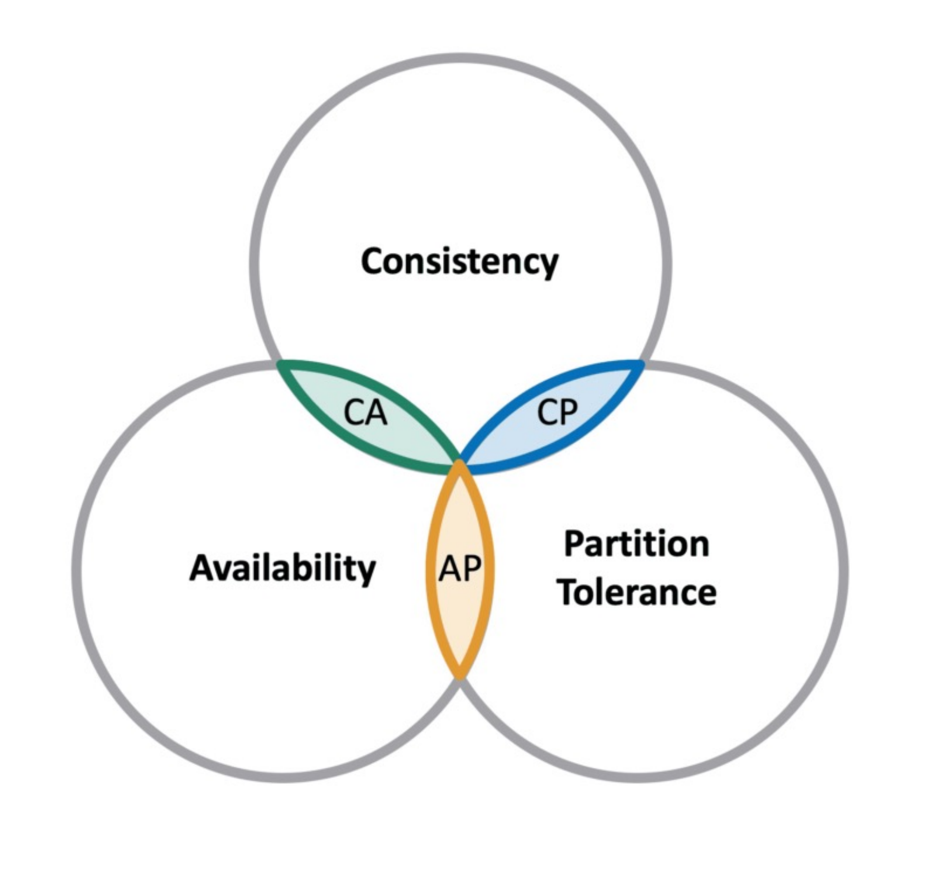
2.4 Database for Serving Data: Transactional Database

- MySQL: RDBMS
- MongoDB: NoSQL, CP, B-tree based, Persistent
- ScyllaDB: NoSQL, AP, LSM-tree based, Persistent
- Redis: NoSQL, In-memory

2.4 Database for Serving Data: Transactional Database

	RDBMS	NOSQL
Schema	Primarily strict and predefined.	Flexible
Implementation	Implementation in the form of normalized tables, using table joins.	Document-based, graph databases, key-value pairs, wide-column stores
ACID	Guarantees	Not typically guaranteed
Examples	MySQL, MariaDB, Oracle, PostgreSQL	MongoDB, Cassandra, DynamoDB, (Redis)

2.4 Database for Serving Data: Transactional Database



- Consistency: All nodes can see the same data at the same time (returning the most recent data).
- Availability: All requests can be successful or failed (reading/writing is always possible without errors).
- Partition-tolerance: The system can continue to operate even if message delivery fails or part of the system (network) breaks down.

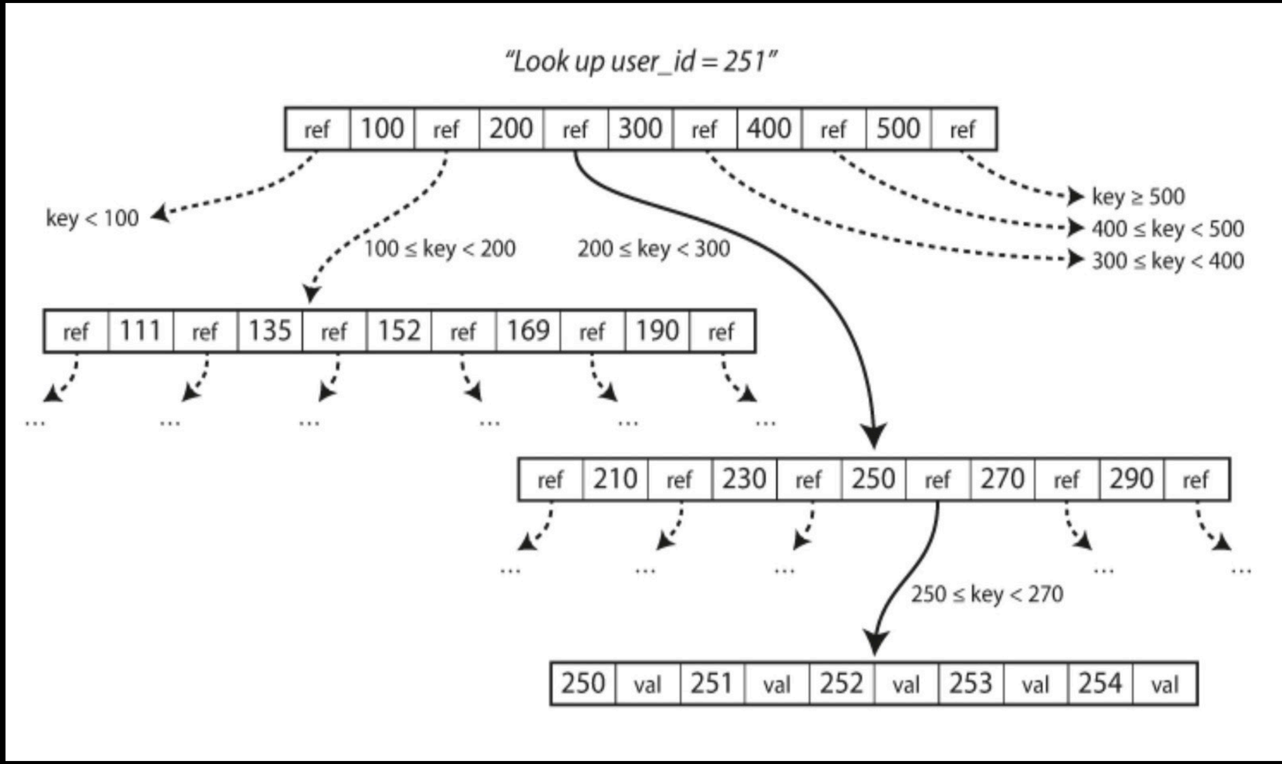
2.4 Database for Serving Data: Transactional Database

B-tree

- Page-oriented: Optimizing node size to match the operating system's page size of 4KB for loading data from disk to memory efficiently.

- WAL (Write-ahead log): Recording logs before insertion operations to enable recovery in case the tree becomes corrupted.

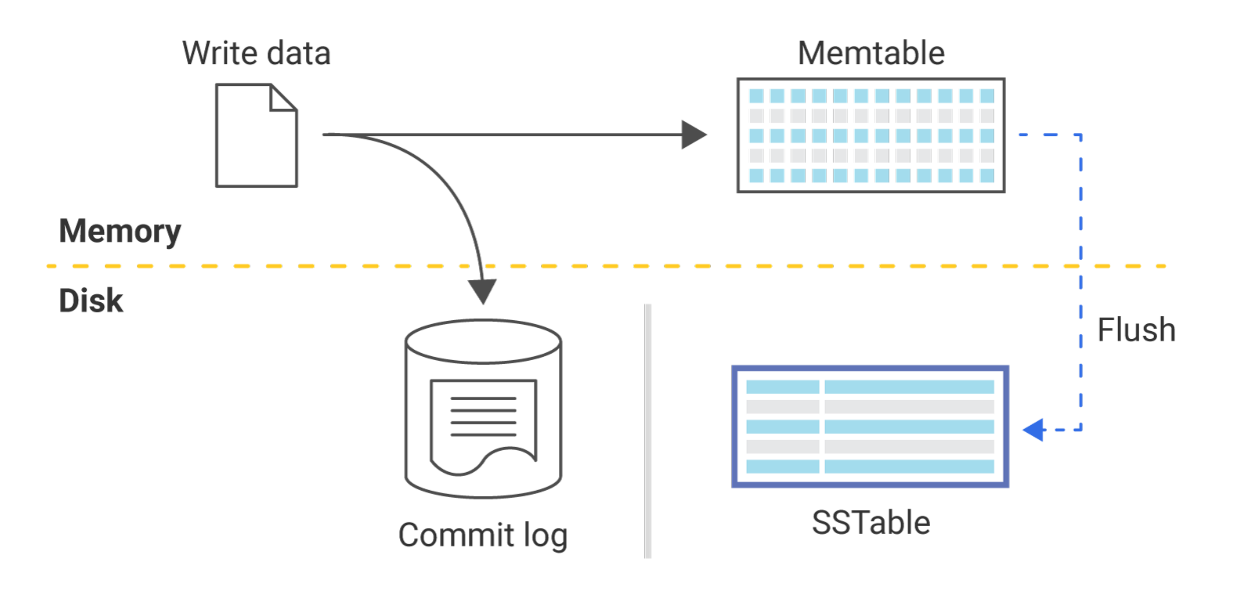
- Fast read, slow write



2.4 Database for Serving Data: Transactional Database

LSM-tree

- SSTable (Sorted String Table): insertion commands are initially stored in a memory cache. Once the cache reaches a certain threshold, the data is batched, sorted, and stored as block-level logs (flush).
- Compaction: Scanning all records is required during searches. To address this issue, LSM-Tree periodically merges SSTables.
- Fast Write, Slow Read



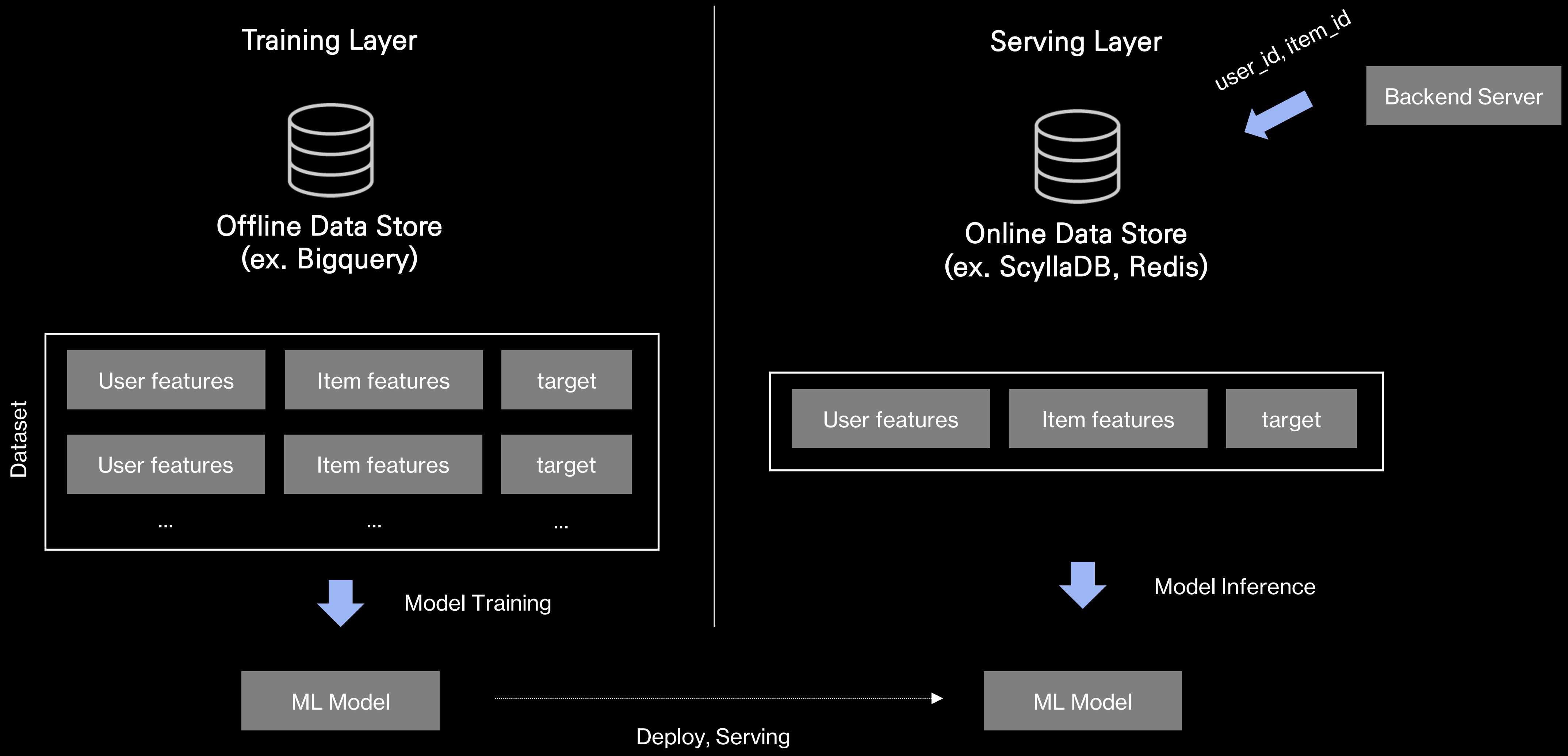
2.4 Database for Serving Data: Transactional Database

In–memory store

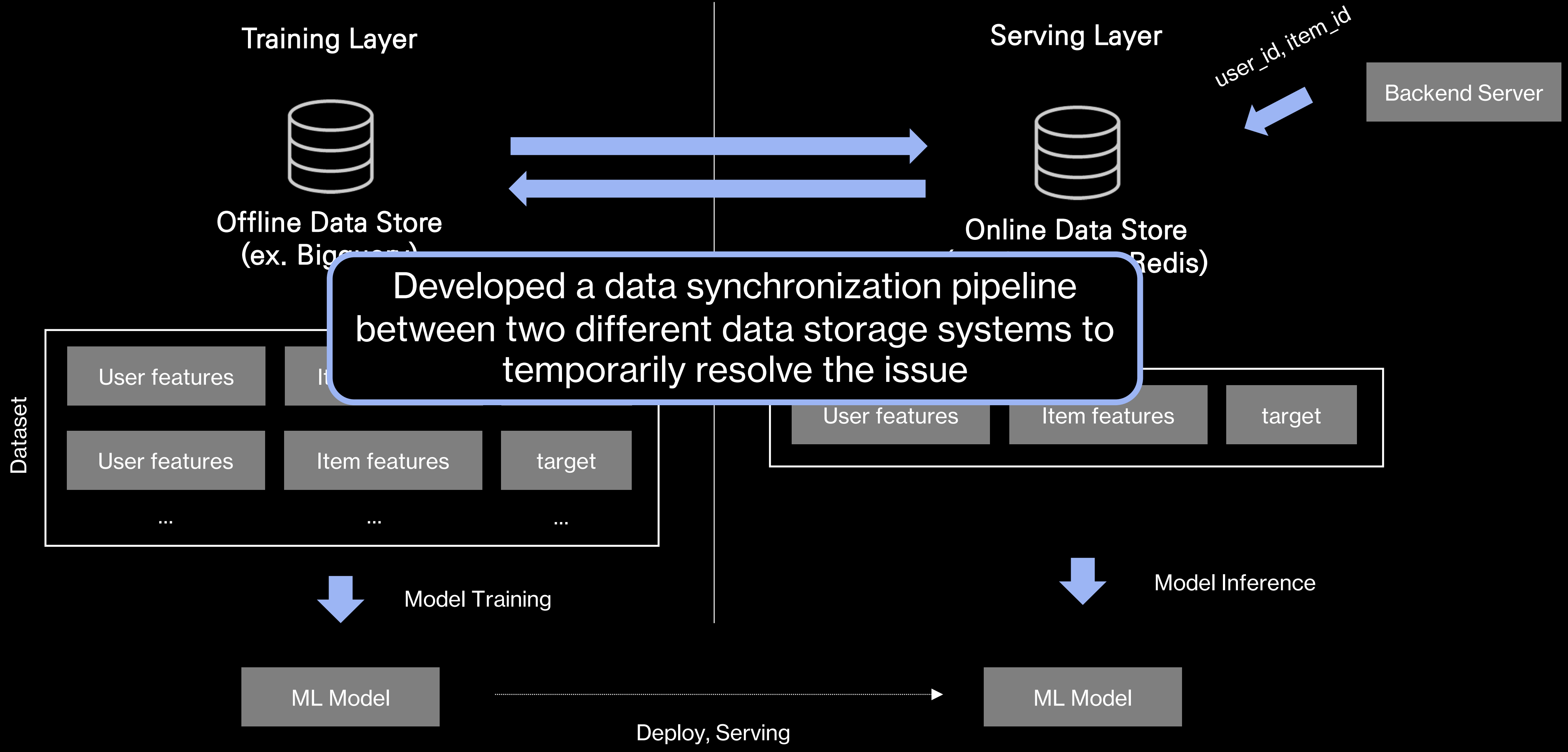
- All data is loaded into memory without using disk.
- Fast write / fast read.
- High cost due to RAM usage and low durability.

3. Why was the Feature Store Introduced?

3.1 Hyperconnect Recommender Systems Before the Feature Store



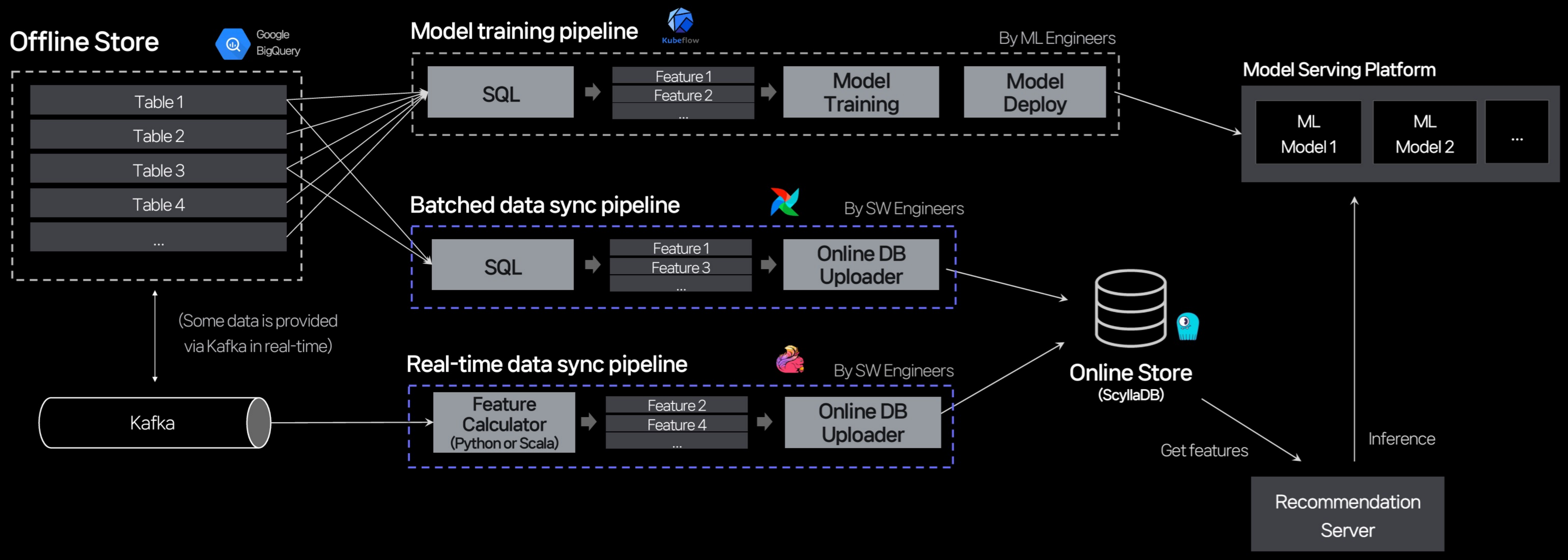
3.1 Hyperconnect Recommender Systems Before the Feature Store



Developed a data synchronization pipeline between two different data storage systems to temporarily resolve the issue

3.1 Hyperconnect Recommender Systems Before the Feature Store

The actual system architecture had the following structure



3.2 Challenges

- When operating a single recommendation system, the absence of a Feature Store didn't pose significant issues.
- However, as we applied recommendation systems in various places, more problems arose

01 Mismatch between training and serving data

02 High engineering costs when adding features

03 Duplication of components when operating multiple recommendation systems

04 Difficulty in sharing features among multiple recommendation systems

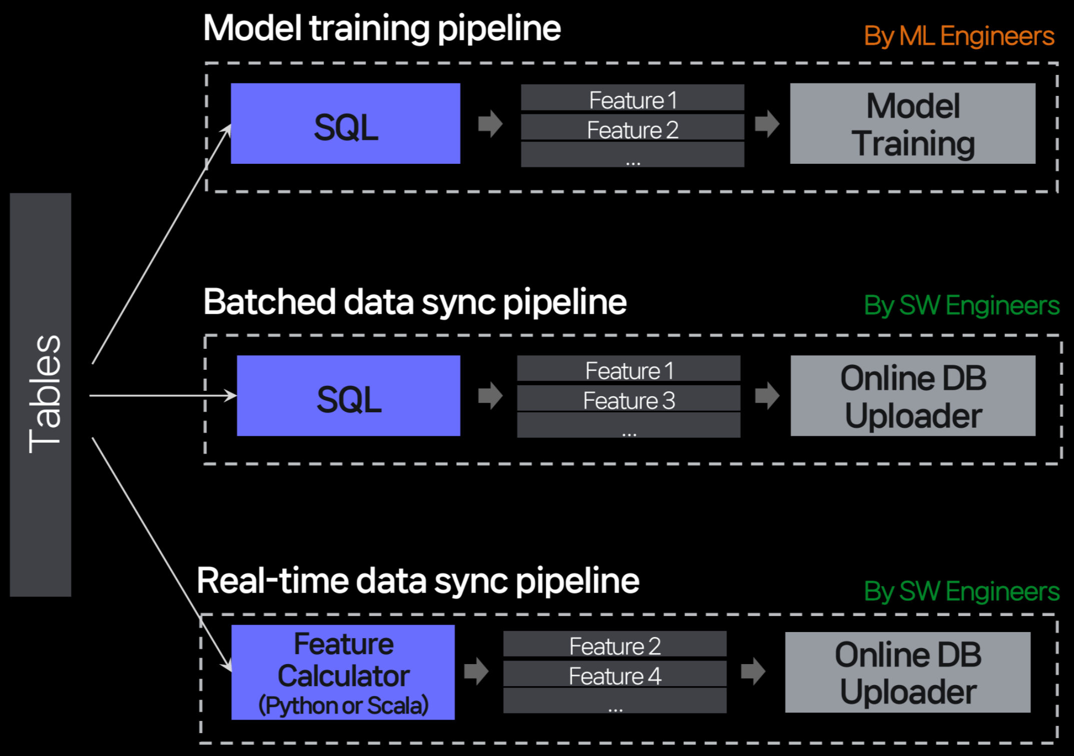
3.2.1 - Mismatch Between Training and Serving Data

Ideal:

When querying features with the same User ID, the same data is returned in both the training and serving layers

Reality:

- The feature calculation logic is divided into three different places, with different engineers for each pipeline.
- This results in data inconsistency between the training and serving layers.
  (ex) the average chat duration or time spent for the same user may differ between BigQuery and the Online DB.)



3.2.2 – High Engineering Costs When Adding Features

The software engineering tasks for adding new features to the model:

1) Modify the schema of the online data storage.

2) Develop data synchronization pipelines for the new features.

3) Backfill the new features into the online storage.

4) Add/modify logic in the backend servers to use the new features.

Problems

Frequent occurrence of slowed iteration speeds in model online experimentation due to bottlenecks caused by software development tasks

3.2.3 – Duplication of Components When Operating Multiple Recommender Systems

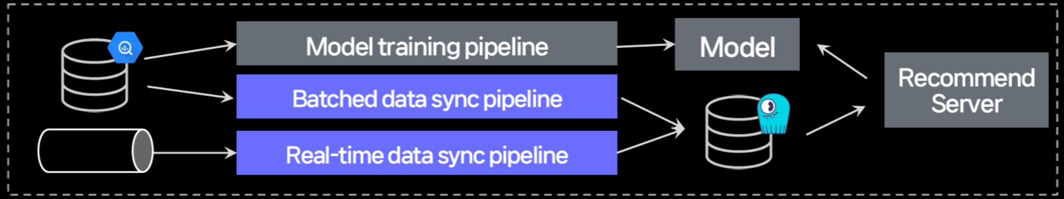
- The model training pipelines varied slightly between recommendation systems, with minimal duplication.

- However, there was significant duplication of logic in the data synchronization pipelines.

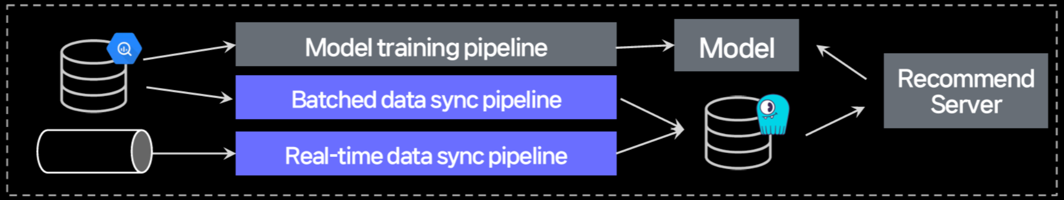
  Ex), offline DB connectors, online DB connectors, data validation logic, parallel execution, incremental update logic, throughput limiter, etc.

- This acted as technical debt whenever a new recommendation system was added.

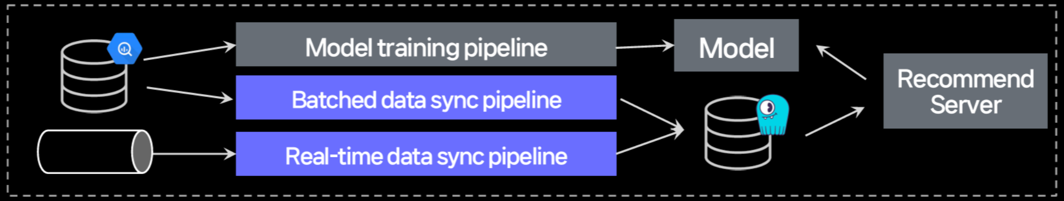
Recommender Systems 1 (ex. Arar 1:1 Matching)



Recommender Systems 2 (ex.Hakuna



Recommender Systems N



3.2.4 – Difficulty in Sharing Features Among Multiple Recommendation Systems

- In a single service (e.g., Azar), there can be multiple types of recommendation systems (e.g., 1:1 Matching, Live, Lounge).
- Even if they share the same user base, the data schema and type of online storage may differ between recommendation systems, making feature sharing difficult.
  - For example, one recommendation system may use MongoDB as the online storage, using a flatten key-value data structure and JSON for serialization.
  - Another recommendation system may use Redis as the online storage, using nested data structures and protobuf for serialization.
  - Yet another recommendation system may use ScyllaDB as the online storage, directly adding columns to the database for serialization.
- This complexity makes it challenging to share features among multiple recommendation systems.

3.3 Reasons for Adopting a Feature Store & Role

Role

- Solving the issue of data inconsistency between training and serving data, and acting as a platform to centralize various components needed for operating multiple recommendation systems.

Reasons

- Centralizing various components that emerged from operating multiple recommendation systems and leveraging technology

4. Feature Store of Hyperconnect

4.1 Open-source? In-house development?

Requirements for the Hyperconnect recommender system

01 Real–time calculation and usage of features

- Features should be calculated and used in near real–time.
- Real–time features have a significant impact on performance, especially when recommending users rather than static items.
- Side–information features should be updated within seconds after user feedback occurs.

02 Support for historical features

- The system serves session–based recommendation models, requiring support for historical features.

03 Support for BigQuery as the offline storage and ScyllaDB (Cassandra compatible) as the online storage

- BigQuery and ScyllaDB are already major technologies used within the company.
- Maintaining the existing stack is efficient for overall infrastructure management.

4.1 Open-source? In-house development?

Decision to develop in–house:

- No open–source solution that fully met our requirements
- comparing the features of the most active open–source project, Feast, with our in–house requirements

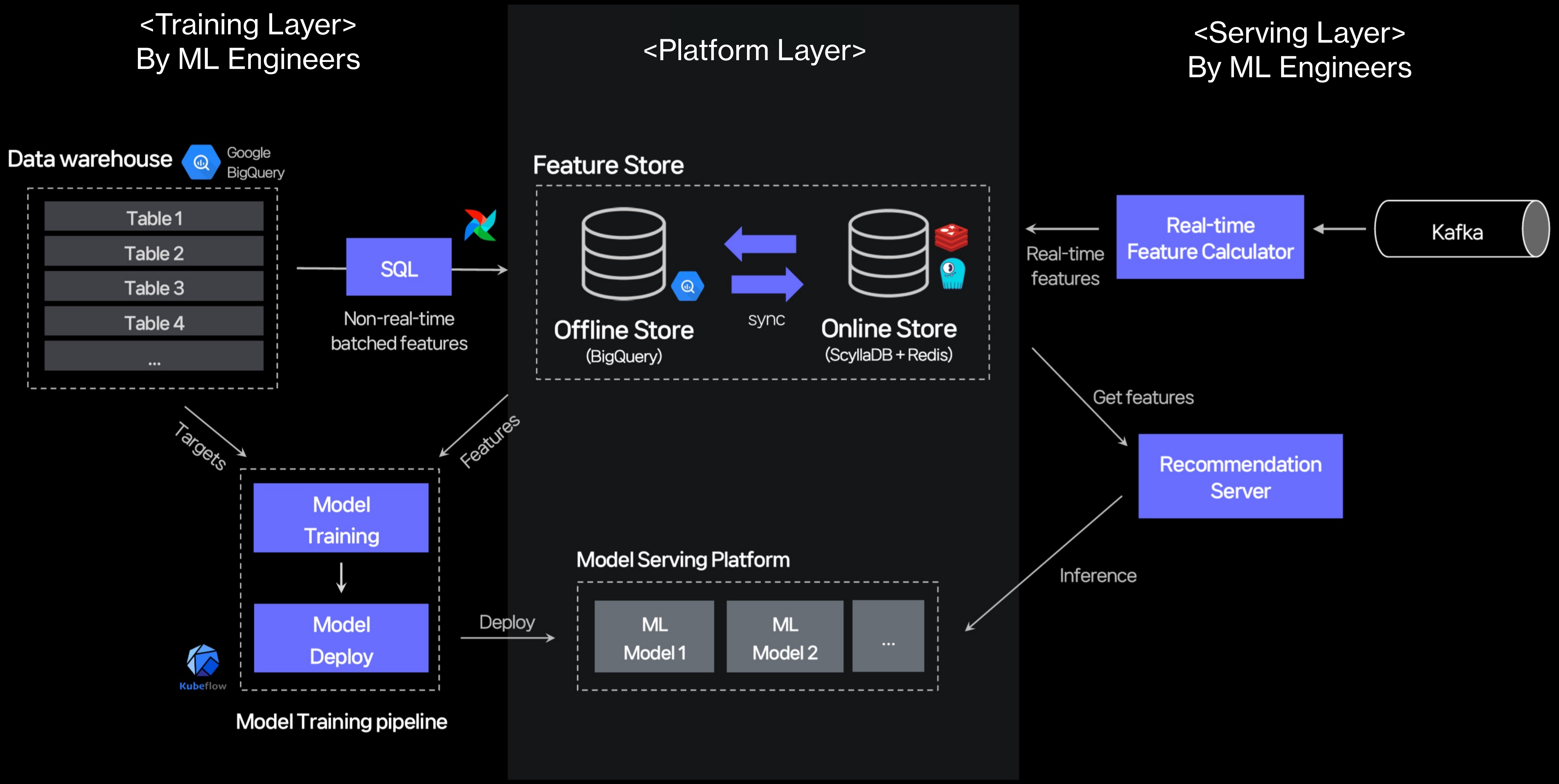
	Feast	In-house Feature Store
Historical feature support	X	O
Offline -> Online data synchronization	O	O
Online -> Offline data synchronization	X	O
Support for real-time updates in online storage	△	O
Support ScyllaDB as an online storage	X	O
Point-in-time Join Support	O	X

4.1 Open-source? In-house development?

Scope of the In-house Feature Store:

- Focus only on solving the most essential problems, since in-house development can require significant resource
- Address the challenge of creating a unified data storage for training/serving recommendation systems!
- Avoid providing additional functionalities like feature discovery or point-in-time join.
  - Feature discovery will continue to be performed using BigQuery as before
  - Point-in-time Join can be implemented either using SQL as previously done or within streaming applications

4.2 After the Feature Store: HyperConnect's recommendation system



4.3 Features of the Hyperconnect Feature Store

01 GitOps-based feature definition system

02 Offline to online storage synchronization pipeline (Upsync)

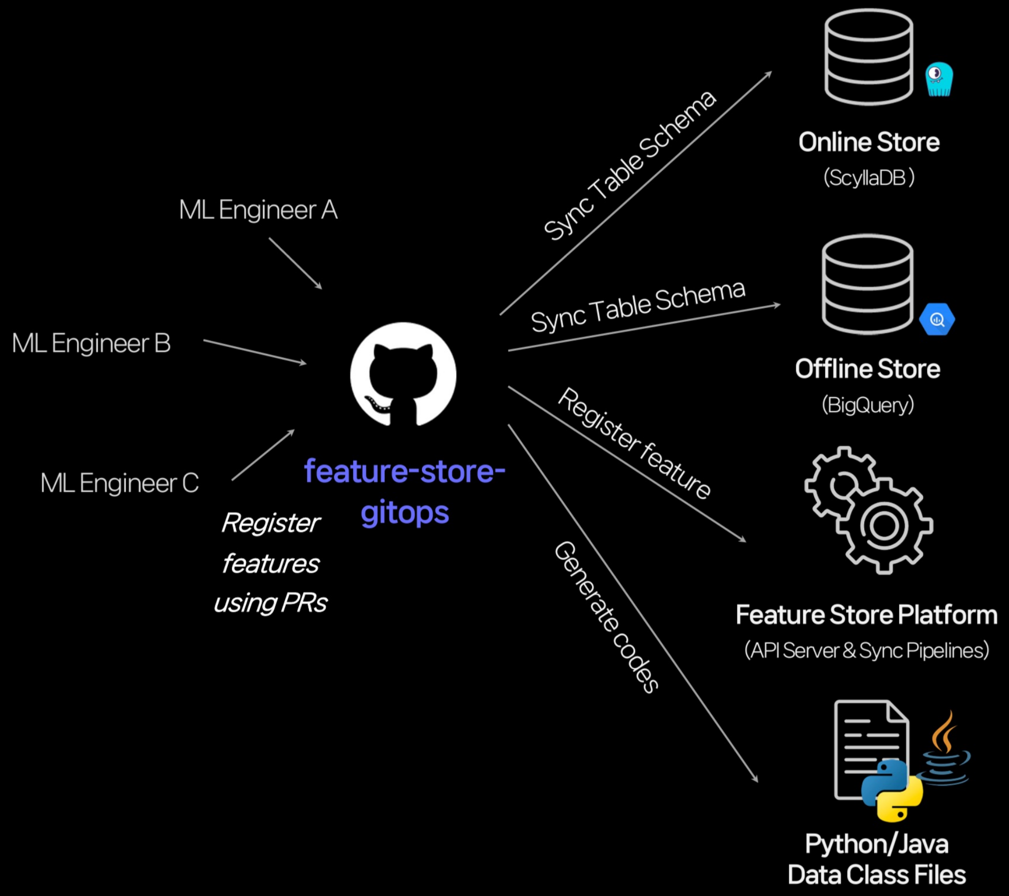
03 Online to offline storage synchronization pipeline (Downsync)

04 Online feature Read API

05 Access control and Data Governance support

4.3.1 GitOps-based feature definition system

Managing specifications for all features using Git and automating various tasks using GitOps




```yaml
live::hakuna-host-stat-v1:
  feature_type: Object
  owner: "shawn.s"
  schema:
    user_id:
      type: Text
      is_key_field: true
    gender:
      type: Text
      description: ""
      default_value: "MALE"
    level:
      type: Long
      description: ""
      default_value: 0
    country_code:
      type: Text
      description: ""
      default_value: "XX"
    birthday:
      type: Text
      description: ""
      default_value: "2000-01-01"
```

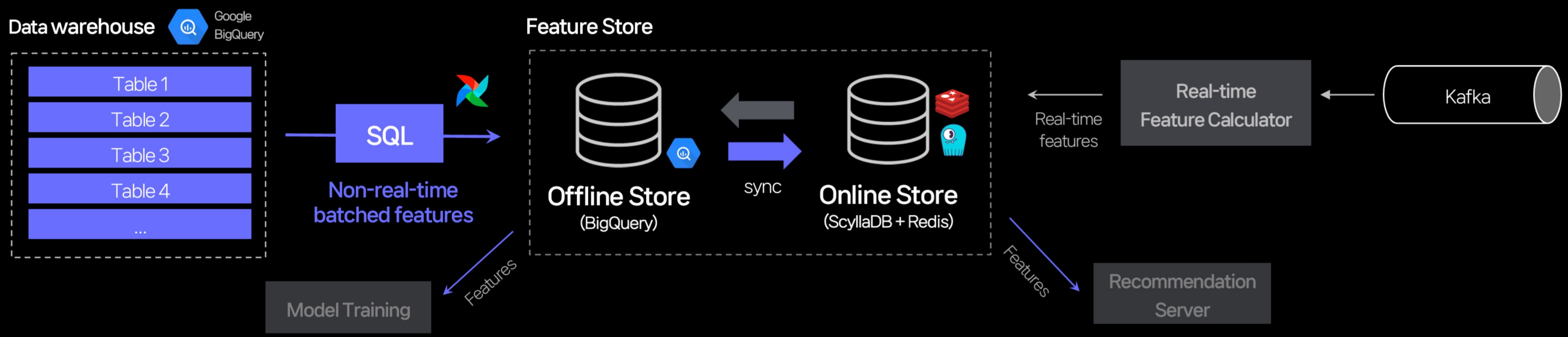
```yaml
azar-match::rich-history:
  feature_type: History
  owner: "ray.l"
  schema:
    user_id:
      type: Long
      description: ""
      is_key_field: true
    user_gender:
      type: Text
      description: ""
      default_value: ""
    user_country_code:
      type: Text
      description: ""
      default_value: ""
    user_language_code:
      type: Text
      description: ""
      default_value: ""
```

4.3.2 – Upsync pipeline (Offline -> Online Store)

- The feature of synchronizing data from the offline storage (BigQuery) to the online storage

- Just write SQL queries and register the pipeline in Airflow, and you can immediately create features that are usable in both offline and online storage.

- It's very convenient to use since only SQL needs to be written, saving software engineering resources. However, there is a limitation that real-time features cannot be used



4.3.2 – Upsync pipeline (Offline -> Online Store)

Just register the GitOps YAML and write the Airflow DAG, and the configuration is complete!

GitOps Yaml

```yaml
azar-lounge::user-interaction:
  feature_type: Object
  owner: "shuki"
  schema:
    ...
  upsync:
    enabled: true
    incremental_update: true
```

Incremental Update: An option that updates only the data that has been updated since the last synchronization point, instead of synchronizing all data every time.

AirFlow DAG

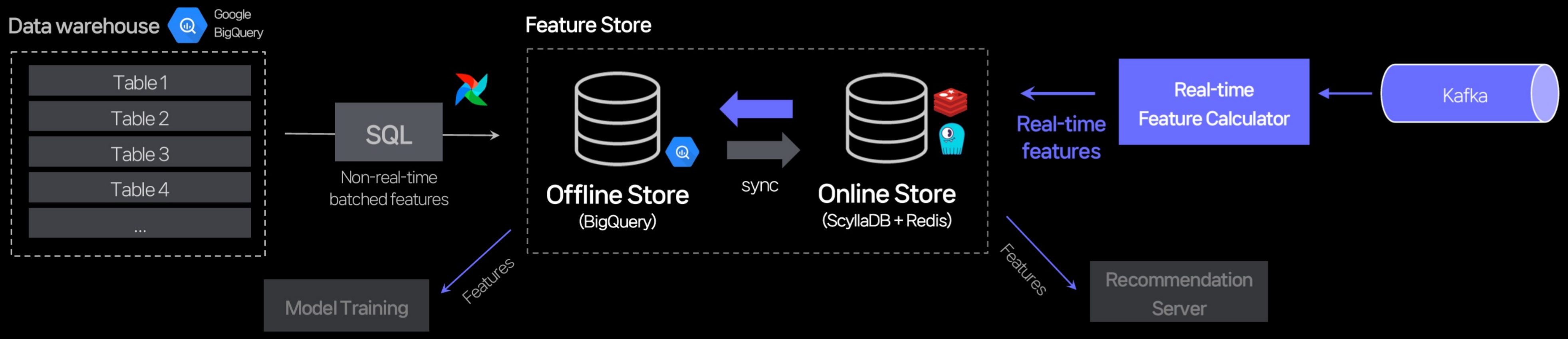
```python
transform_user_stats = BigQueryOperator(
    task_id='transform_user_stats',
    use_legacy_sql=False,
    sql=open(f"{dirname}/query/transform/user_stats.sql").read(),
    dag=dag,
)

feature_store_upsync_worker_op = FeatureStoreUpsyncOperator(
    target_features=[
        "azar-lounge::user-interaction",
        "azar-lounge::user-stats",
    ],
)

[
    transform_user_interaction,
    transform_user_stats
] >> feature_store_upsync_worker_op >> end
```

4.3.3 – Downsync pipeline (Online -> Offline Store)

- The feature of calculating features in real-time on the online backend servers and registering them in the Feature Store, then synchronizing them to the offline storage (BigQuery), similar to Change Data Capture (CDC)
- This functionality requires more software engineering resources compared to Upsync, but it is useful for models where real-time performance is crucial
- For example, in live streaming recommendations, features like real-time viewership, vision features, and click rate features for new sign-ups



4.3.3 – Downsync pipeline (Online -> Offline Store)

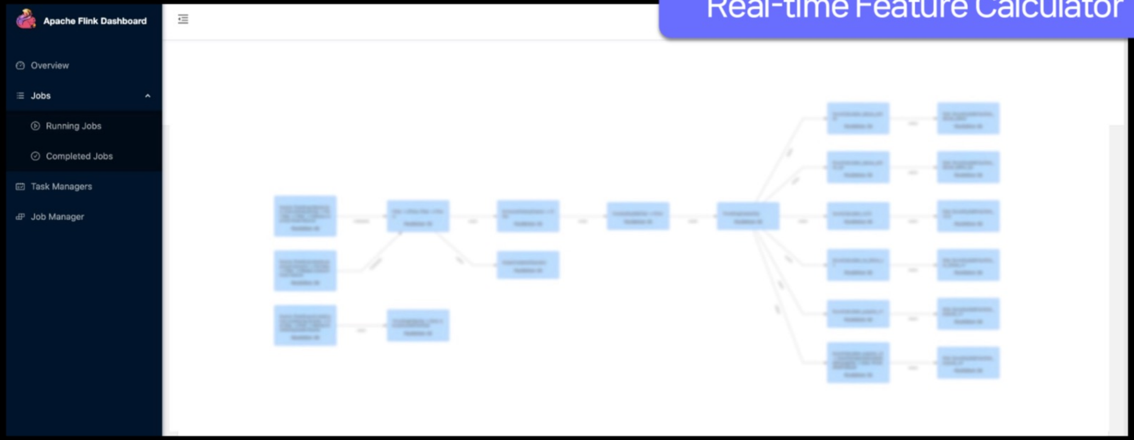
Mainly using event streaming applications like Apache Flink for real-time feature calculation.
Feature updates are performed by sending commands to Kafka.

GitOps Yaml

```yaml
live::realtime-room-context:
  feature_type: Object
  owner: "owen.l"
  schema:
    ...

  downsync:
    enabled: true
    min_interval_by_key: "1m"
    random_sampling_ratio: 0.9
```

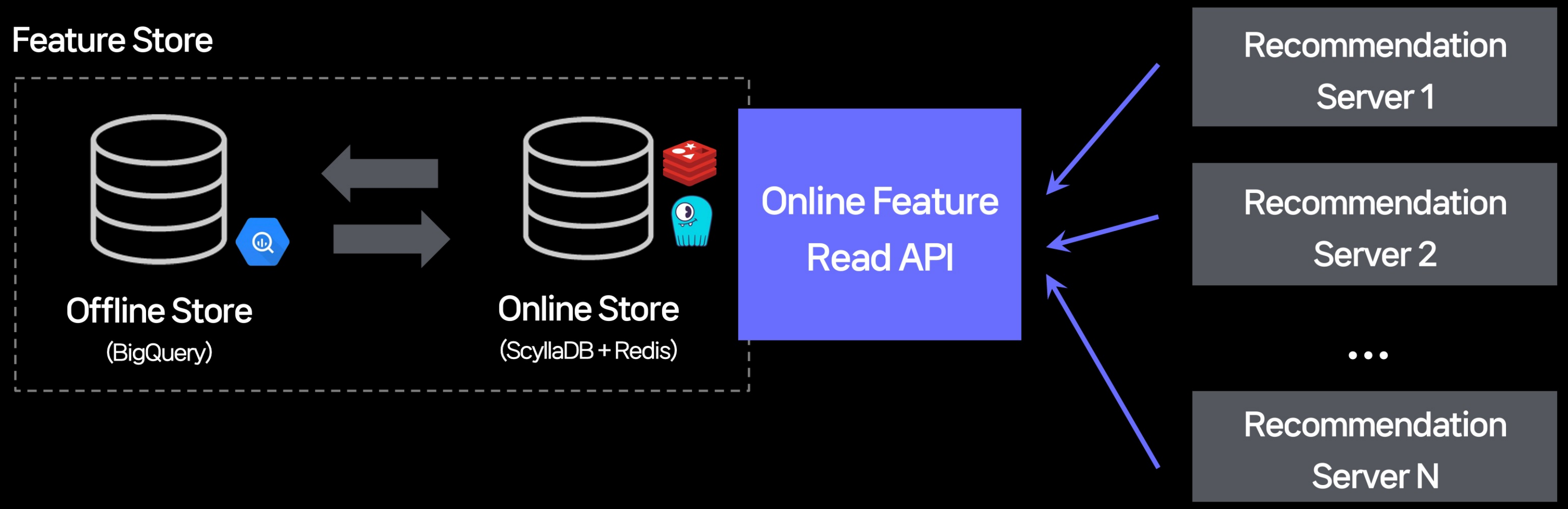
Real-time Feature Calculator



*Min Interval By Key 및 Random Sampling Ratio Update:
Options for which sampling policy to use when synchronizing feature updates to the offline storage

4.3.4 – Online Feature Read API

- The recommendation servers access features through the Read API instead of directly accessing the online data storage
- API server supports access control, deserialization (Avro), and caching options (Redis).
- Initially developed and operated with FastAPI, but migrated to Spring due to performance issues
- TPS: Several thousand or more / p99 latency: 25ms



4.3.5 – Access Control & Data Governance Support

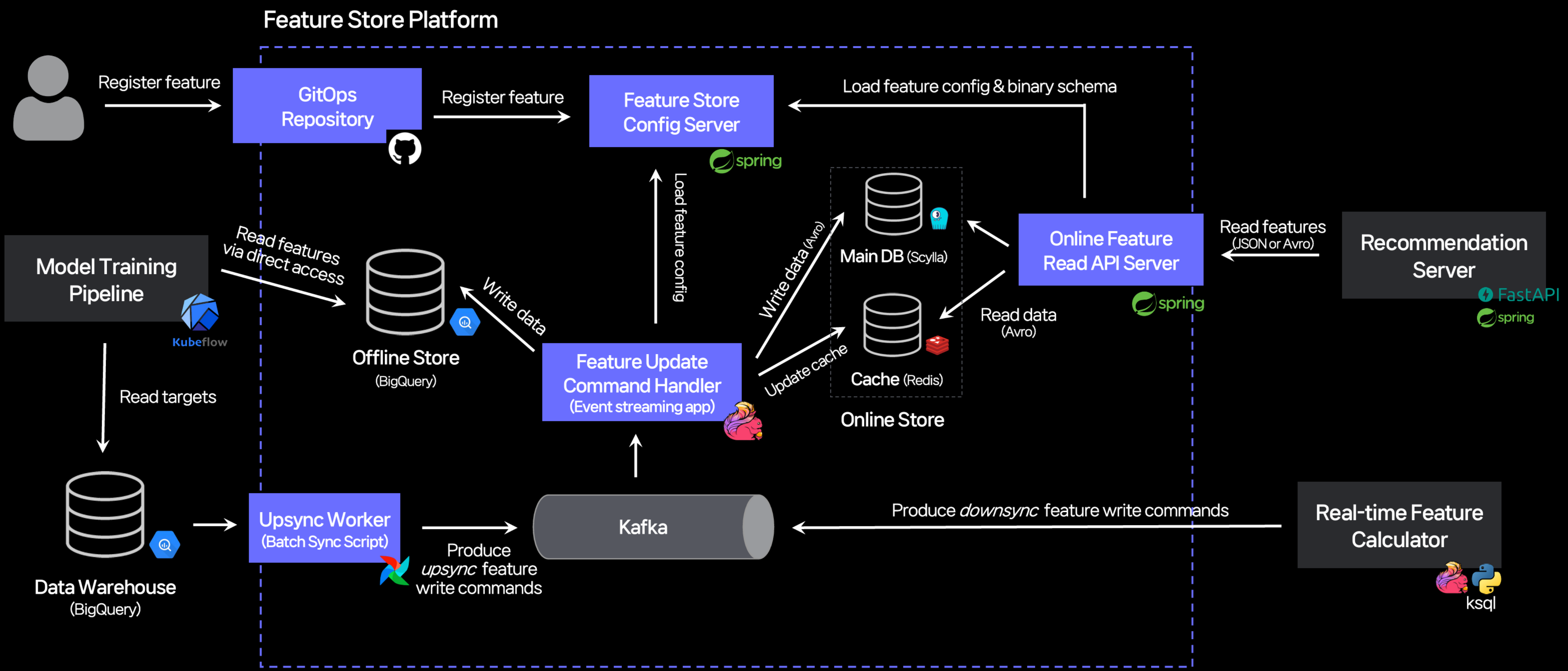
Access Control

- Online Read API: it's possible to set accessible tables for each microservice/developer.
- Offline Storage: Use Bigquery's access control.

Data Governance

- Data governance is being managed through data retention in both offline and online storage.
- Since the retention period varies for each business, we provide the ability to set retention periods for each feature.

4.4 Hyperconnect Feature Store Internal Architecture



4.5 Summary of Usage of the Hyperconnect Feature Store

Feature Definition	Register with GitOps
Write Feature	Training layer: write SQL and use Airflow Serving layer: use Kafka
Sychronization between storages	Offline -> Online: Activate Upsync pipeline Online -> Offline: Activate Downsync pipeline Activation method: Just add a flag in GitOps and you're done!"
Read Feature	Training layer: use Bigquery Serving Layer: through Read API server

5. Case Studies & Adoption Impacts

5.1 Use Cases

Services where the Feature Store is applied

- he majority of existing recommendation systems, including those within the Azar and Hakuna services, with over 5 recommendation systems integrated.
- All newly started recommendation systems also adopt the Feature Store.
- The Feature Store is also utilized in anomaly user detection systems, in addition to recommendation systems.

5.1 Use Cases

Types of recommendation systems where the Feature Store is applied

- Boosting-based CTR (Click Through Rate) prediction models.
- Deep learning-based time spent prediction models.
- Session-based recommendation systems that utilize real-time history information.
- Recommendation systems that extract vision information from real-time videos and use it as input for the model (limited to live streaming).

5.2 Resolution of data consistency issues

Before Adoption

- Data inconsistency issues discovered approximately once a month.
- Significant discrepancies found between feature statistics analyzed in BigQuery and the actual feature statistics being used as inputs for the models.

After Adoption

- No data inconsistency issues discovered after the introduction of the Feature Store.

5.3 Development productivity

The time taken to use new features in the serving layer

(after feature engineering and modeling work is completed, until new model experiments)

Using only batch features

Using real–time features too

5.3 Development productivity

The benefits from the perspective of ML engineers

1. Reduced communication costs with software engineers.
2. Ability to reuse features created once in multiple recommendation models.
3. Ability to reuse features created by other ML engineers in the recommendation models I create.

5.3 Development productivity

The benefits from the perspective of software engineers

1. Time saved on developing feature synchronization pipelines and logic, enabling focus on core logic development

2. No need to worry about the issue of training/serving data inconsistency

5.4 The effects of platformization

Most recommendation systems utilize the Online Feature Read API structure.
–〉 With just one feature addition, numerous recommendation systems can benefit simultaneously.

1. By integrating Redis Cache, we observed a simultaneous reduction in latency across recommendation servers.

2. Additionally, with the adoption of binary serialization (Avro), we were able to save on data storage costs and achieve a reduction in network latency, with some features compressed by over 80%.

3. Upon the introduction of anomaly detection systems, we anticipate widespread benefits across numerous recommendation systems.

5.5 Challenges In The Adoption Process

- To apply the Feature Store effectively, I directly integrated it with multiple recommendation systems.
- There were numerous internal Feature Store presentations, conducted separately for ML engineers and software engineers.
- The online Read API server was initially developed using Python + FastAPI, but due to unsatisfactory performance (latency + throughput), it was rewritten in Kotlin + Spring.
- Additionally, as one of the largest clients for the internal shared distributed database (Scylla), I frequently consulted with the DevOps team

Reference

1. 실시간 추천 시스템을 위한 Feature Store 구현기
https://deview.kr/2023/sessions/536

2. 머신러닝 어플리케이션을 위한 데이터 저장소 기술
https://hyperconnect.github.io/2022/07/11/data-stores-for-ml-apps.html

Q&A

Thank you!