Reddit Metadata Store

Reddit's media metadata was distributed across different storage systems. To make this easier to manage, the engineering team wanted to create a unified system for managing all this data

For example, media data used for traditional image and video posts is stored alongside other post data, whereas media data related to chats and other types of posts is stored in an entirely different database..

The high-level design goals were

- **Single System** - they needed a *single* system that could store all of Reddit's media metadata. Reddit's growing quickly, so this system needs to be highly scalable. At the current rate of growth, Reddit expects the size of their media metadata to be 50 terabytes by 2030.
- **Read Heavy Workload** - This data store will have a *very* read-heavy workload. It needs to handle over 100k reads *per second* with latency less than 50 ms.
- **Support Writes** - The data store should also support data creation/updates. However these requests have *significantly* lower traffic than reads and Reddit can tolerate higher latency for this.

Picking the Database

To build this media metadata store, Reddit considered two choices: Sharded Postgres vs. Cassandra.

Picking Postgres

After evaluating both choices extensively, Reddit decided to go with Postgres.
The reasons why included:

- **Challenges with Managing Cassandra** - They found some challenges with managing Cassandra. Ad-hoc queries for debugging and visibility were far more difficult compared to Postgres.

- **Data Denormalization Issues with Cassandra** - In Cassandra, data is typically denormalized and stored in a way to optimize specific queries (*this is based on your specific application*). However, this can lead to challenges when creating new queries that your data hasn't been specifically modeled for.

Data Migration

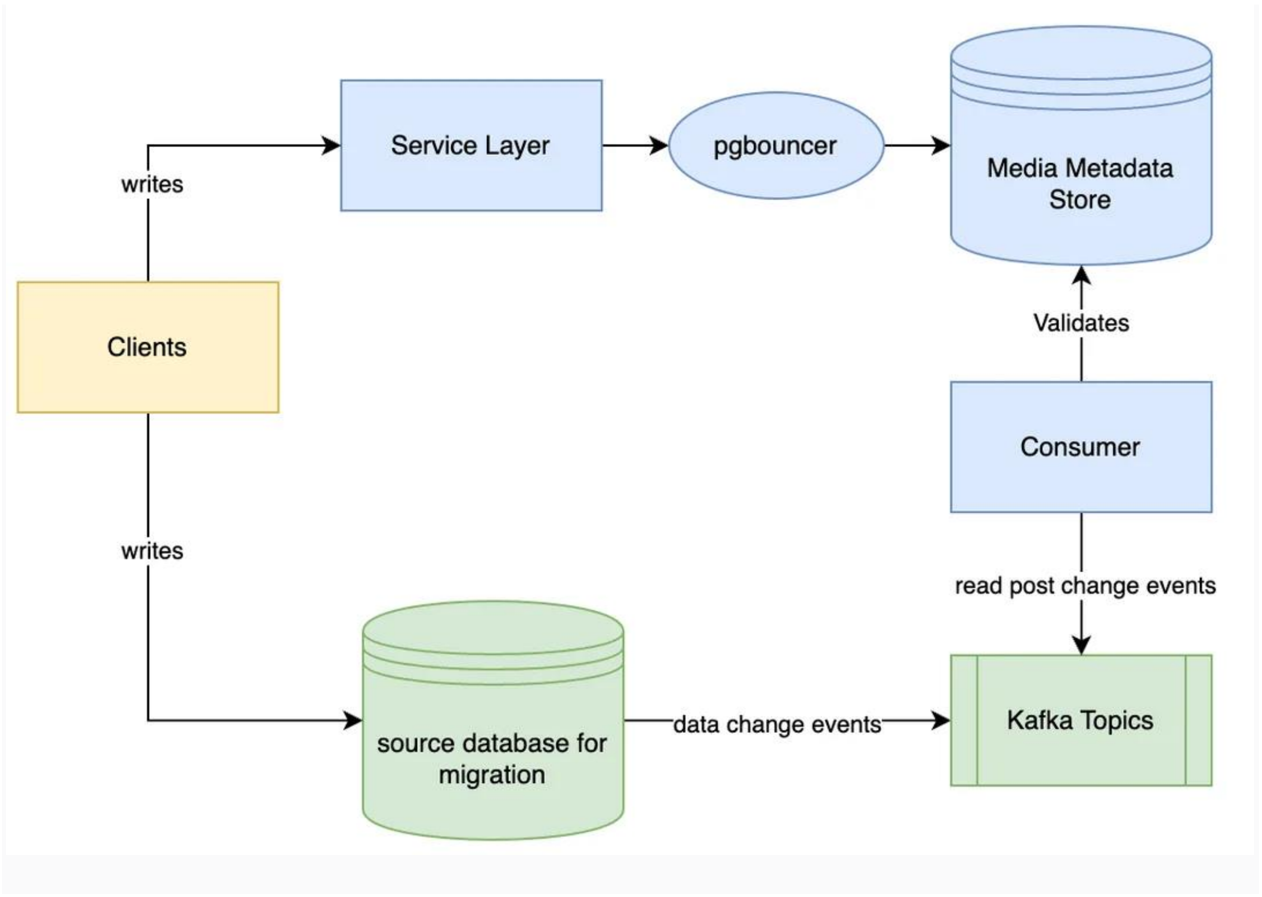
Migrating to Postgres was a big challenge for the team. They had to transfer terabytes of data from the different systems to Postgres while ensuring that the legacy systems could continue serving over 100k reads per second.
Here's the steps they went through for the migration

1. **Dual Writes** - Any new media metadata would be written to both the old systems and to Postgres.
2. **Backfill Data** - Data from the older systems would be backfilled into Postgres
3. **Dual Reads** - After Postgres has enough data, enable dual reads so that read requests are served by *both* Postgres and the old system
4. **Monitoring and Ramp Up** - Compare the results from the dual reads and fix any data gaps. Slowly ramp up traffic to Postgres until they could fully cutover.

There are several scenarios where data differences may arise between the new database and the source:

- **Data transformation bugs in the service layer. This could easily happen when the underlying data schema changes**
- **Writes into the new media metadata store could fail, while writes into the source database succeed**
- **Race condition when data from the backfill process in step 2 overwrites newer data from service writes in step 1**

We addressed this challenge by setting up a **Kafka consumer to listen to a stream of data change events from the source database**. The consumer then performs **data validation** with the media metadata store. If any data inconsistencies are detected, the consumer reports the differences to another data table in the database. This allows engineers to query and analyze the data issues.



Scaling Strategies

With that strategy, Reddit was able to successfully migrate over to Postgres. Currently, they're seeing peak loads of ~100k reads *per second*. At that load, the latency numbers they're seeing with Postgres are

- **2.6 ms P50** - 50% of requests have a latency lower than 2.6 milliseconds
- **4.7 ms P90** - 90% of requests have a latency lower than 4.7 milliseconds
- **17 ms P99** - 99% of requests have a latency lower than 17 milliseconds

They're able to achieve this *without* needing a read-through cache

Table Partitioning

To address this scalability challenge, we have implemented table partitioning in Postgres.

```
SELECT partman.create_parent(
    p_parent_table => 'public.media_post_attributes',
    p_control => 'post_id',      // partition on the post_id column
    p_type => 'native',          // use postgres’s built-in partition
    p_interval => '90000000',    // 1 partition for every 90000000 ids
    p_premake => 30              // create 30 partitions in advance
);
```

We opted to implement range-based partitioning for the partition key `post_id` instead of hash-based partitioning. Given that *post_id* increases monotonically with time, range-based partitioning allows us to partition the table by distinct time periods. This approach offers several important advantages:

Firstly, most read operations target posts created within a recent time period. This characteristic allows the Postgres engine to cache the indexes of the most recent partitions in its shared buffer pool, thereby minimizing disk I/O. With a small number of hot partitions, the hot working set remains in memory, enhancing query performance.

Secondly, many read requests involve batch queries on multiple post IDs from the same time period. As a result, we are more likely to retrieve all the required data from a single partition rather than multiple partitions, further optimizing query execution.