medium.com

Application Design Considerations for Sharding High Volume Databases

Tripti Shastri

18-23 minutes

Tripti Shastri, Jay Patel, Amit Athavale and Srilatha Koneru

PayPal has witnessed an impressive growth in number of customers and transaction processing volumes over the last decade. As we continue to innovate our technology stack and mobile experience, we are tackling several challenges. Scaling our user database (db) is one such challenge that is compounded in its complexity because of our monolithic codebase. The user database employs a traditional relational database at its core. As one can imagine, this database serves several critical business needs across the entire PayPal stack. So, scaling this database while keeping the existing business running is no small feat. This database serves multi-billion queries daily.

For us, the scaling requirements were evident from day one. Whatever solution we came up with had to meet the constraints of strong consistency, high availability, parity with existing business cases along with cost and operational efficiency. We considered multiple strategies to tackle this problem including redirection of reads to replicas and vertical scaling before we started on the path to sharding. There are two keys aspects in sharding, first involves design and making the application shard-ready and the second part involves the actual data movement into new shards. This articles focuses on the first aspect. In this article, we describe the key design choices and the technical challenges associated with the process. We had earlier published an article with learnings around execution. We hope our insights would be helpful to other organizations in their scaling journey.

Sharding

Sharding is commonly used to scale databases. In this technique, data is partitioned into multiple smaller databases, known as shards. Each shard holds a subset of the database and can be accessed by its shard key. Sharding offers a scalable solution while reducing overall hardware costs. Increase in transaction volume is easily handled by adding commodity servers. As such, the cost grows linearly with the increase in traffic volume.

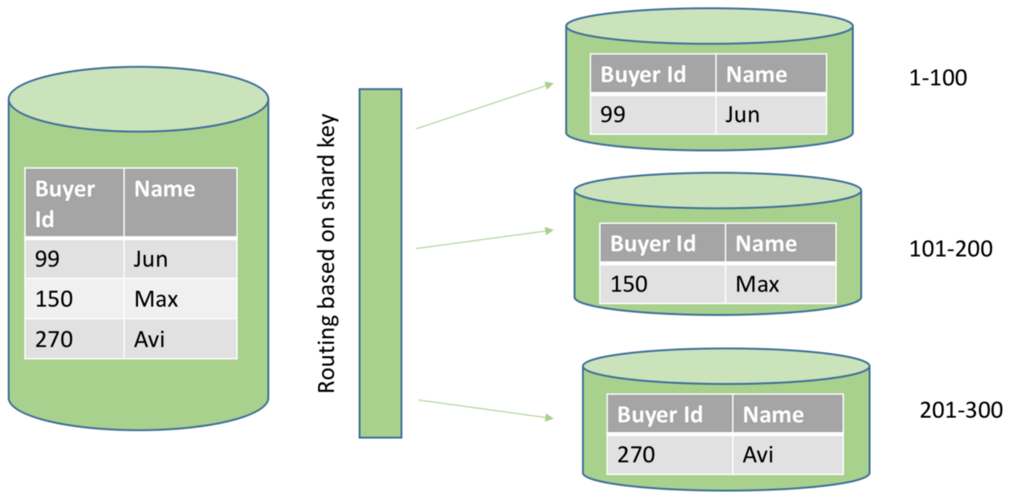


Figure 1. Example of Sharding based on shard key ranges

The above figure represents a simple sharding scheme where rows are partitioned based on the shard key ranges.

Shard Key Selection

Shard key (or partition key) is a primary key which is used to route the queries to appropriate shards. Picking the right shard key is a complex decision. For instance, in the above diagram *buyer id* is used as the shard key. Here are several factors that weigh into this decision:

1. What are the dominant entities in the data? And, specifically, are there any obvious clustering patterns?
2. How much of a schema redesign is needed based on the newly picked shard key?
3. Does the new shard key support the existing query patterns and business use cases?
4. Does it provide flexibility to support future use cases?
5. How it impacts query performance?

> As we examined our data, we observed that the data was clustered around two key entities. The majority of the queries also formed around the same entities. This helped us to establish two shard keys against which the User database is sharded. These two keys are independent of each other. The key insight here is to spend time studying the data and existing application logic up front, which in turn will help in making the right design decisions.

Data Distribution Scheme

A Data distribution scheme determines how the data is partitioned into multiple shards. For instance, in the above figure a simple range-based scheme is used to partition the data in different databases. There are many data distribution schemes available, each one with its own merits and drawbacks. Before choosing the scheme, it is important to establish the key requirements. Here are the requirements we considered:

1. The solution must be scalable to support future growth.
2. Support availability goals.
3. It must be backward compatible.

4. Facilitate easy removal and addition of shard

5. Avoidance of hot spots

6. Minimal data rebalancing

7. Support fast failover

8. Avoid scatter-gather approach for latency sensitive queries

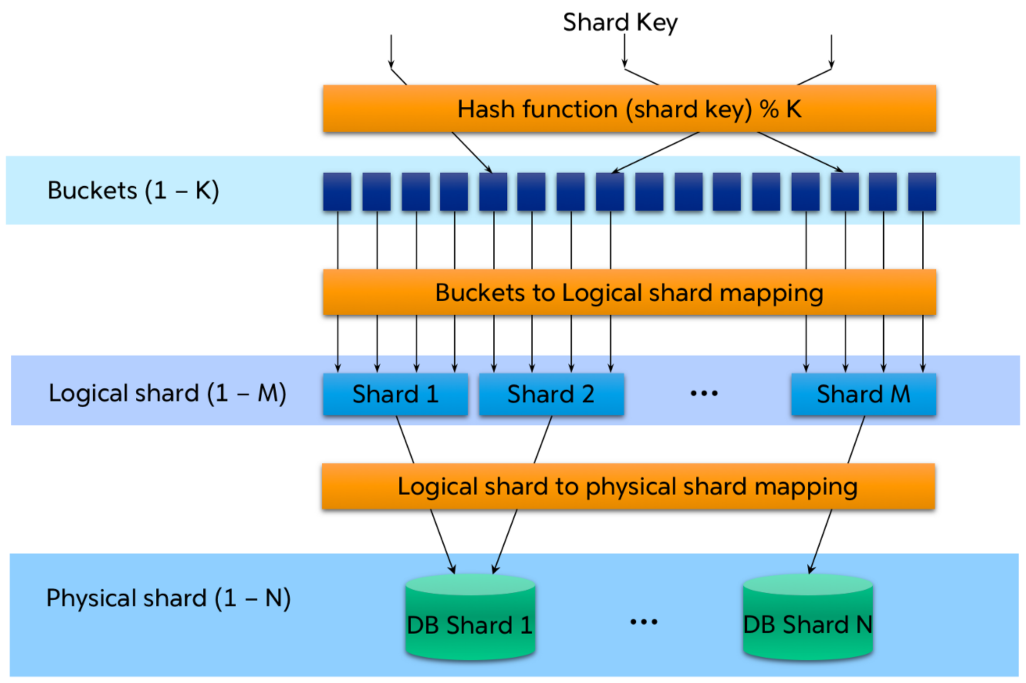


Figure 2. The data Distribution Scheme (Courtesy Kenneth Kang)

We employ an algorithm based technique, where the location of a shard is determined by applying a predetermined algorithm, a hashing function in our case. As shown in figure 2, instead of a directly mapping a shard key to a physical host, shard keys are mapped to a unique bucket. Each bucket is then mapped to a logical host which is then mapped to a physical host. Buckets provide an important abstraction layer, it facilitates addition and deletion of shards and make the entire design more scalable. When a new database is added, it simply involves updating the mapping of buckets (scuttles) to logical shards (assuming the new db host has the necessary data copied).

Transformations to Enable Sharding

A prerequisite to sharding is ensuring the application accessing the database is shard-ready, meaning that all access is via the shard key. The process of making an application shard-ready involves a series of transformations to the underlying data model and interfaces. One has to ensure that following specifications are met by both the data model and the application layer.

Data Model

1. Each table in a sharded database must have a shard key column.
2. The id generation process must guarantee uniqueness across all the shards.
3. Tables might need denormalization or other forms of transformations to support 1 and 2 below.

Application Layer

1. Join across tables must be achieved using the same shard key. Since, the records are partitioned based on a shard key, different shard keys may reside in different databases.
2. No write should span across multiple shards.
3. The read interfaces must be shard key based.

There are unique design challenges to address for each of the above steps. We will cover them in the rest of the article.

Application Layer Design Considerations

Transactions

“A transaction is a collection of read/write operations succeeding only if all contained operations succeed. Inherently a transaction is characterized by four properties (commonly referred as ACID): Atomicity, Consistency, Isolation and Durability”[3]. The changes required at the application layer mostly stem from how transactions are designed in an application. At a very basic level each query could be considered a separate transaction (auto-commit) or could involve multiple queries involving several tables. In a sharded environment, distribution of data across databases pose challenges to the existing transactions. Before dwelling into how these challenges are addressed, lets first establish what is a transaction boundary or how do we determine what a transaction entails? To help answer this question, we first define a transactional entity. Transactional entity is grouping of tables that participate in a transaction. Transactional entities have following properties:

1. Tables belonging to the same entity are sharded based on the same shard key.
2. For a given entity, all tables could be joined or these tables could be part of the same transaction with the same shard key.

Consider an online store where the entities involved are customer, order, inventory, phone, address, email, shipment, and so on. Assuming customer-id to be the shard key, a coarse grain entity would be customer; fine grain entities would be phone, address, etc. Each type of the entity is a possible candidate for a transactional entity. Following are the pros and cons of choosing a fine grain vs coarse grain entity as a transactional entity:

1. Fine grain entity as a transactional entity provides more flexibility and allows us to create individual shards for each. However, on the flip side, it may lead to breaking transactions and queries impacting consistency and performance. In a large scale system, it also adds maintenance overhead.
2. Course grain entities provide better performance and consistency as they allow joins across different tables and entities. Transactions spanning beyond a single entity need not be broken. For our use case, we chose this option.

Join Across Shards

A common example where application layer joins are required are "IN" clause queries. For example, *select x where customer-id in (1,2,3,…)*. As individual shard keys can reside on different databases, join across shards is not possible. So the only alternative is to separate queries and perform joins in the application code. If it turns out that an application requires lot of joins to serve data, it most likely is symptomatic of either a sub-optimal shard key or that the data model requires tuning.

Handling Failures - In case of a failure to retrieve data from one or more shard, one can choose to either fail the entire query or return partial data. The former impacts availability but provides most accurate data. The latter approach compromises on accuracy to favor availability.

Write Across Shards

Since writing across shards is not a possibility, we had to break any transaction that spanned across different shards. This is a relatively complex task as we have to account for impact on consistency, business rule violations and possible deadlock scenarios. Here are some common examples where breaking a transaction would be necessary.

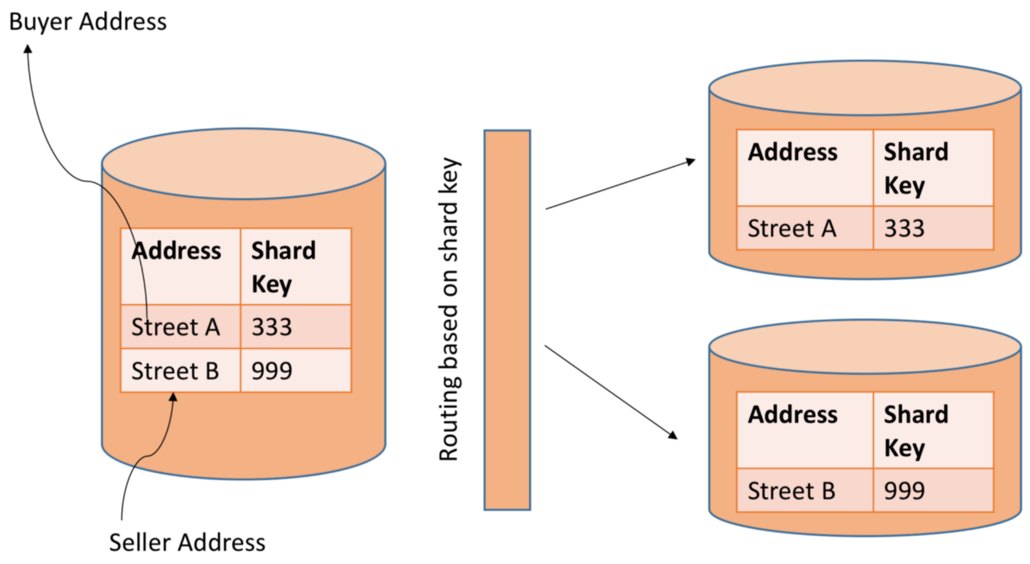


Figure 3. Two records belonging to the same table are placed in different shards.

1. Writes involving different shard keys but the same tables as shown in figure figure 3. In this scenario, a transaction contains both shipping address for a buyer and warehouse address for a seller. If these addresses are placed on different shards, this can lead to write across shard scenario.

2. Write involves different entities sharded by different shard keys as shown in figure 4. In this scenario a transaction encompasses both Inventory entity and Order entity. These two entities have different shard keys and reside on different databases after sharding. This is a violation of the transactional entity properties and requires refactoring of the transaction.

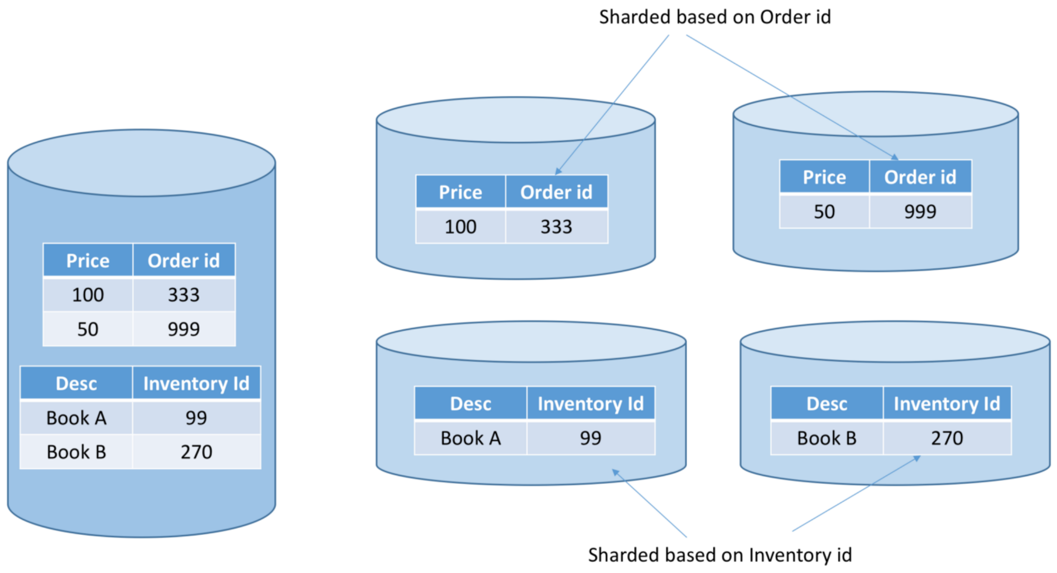


Figure 4. Two different entities sharded by different shard keys

Handling Failures - When a transaction is broken into multiple transactions, we have to account for the possibility that one of the transactions could fail. In such scenario we may have only partial data in the database affecting consistency. The possible solutions are,

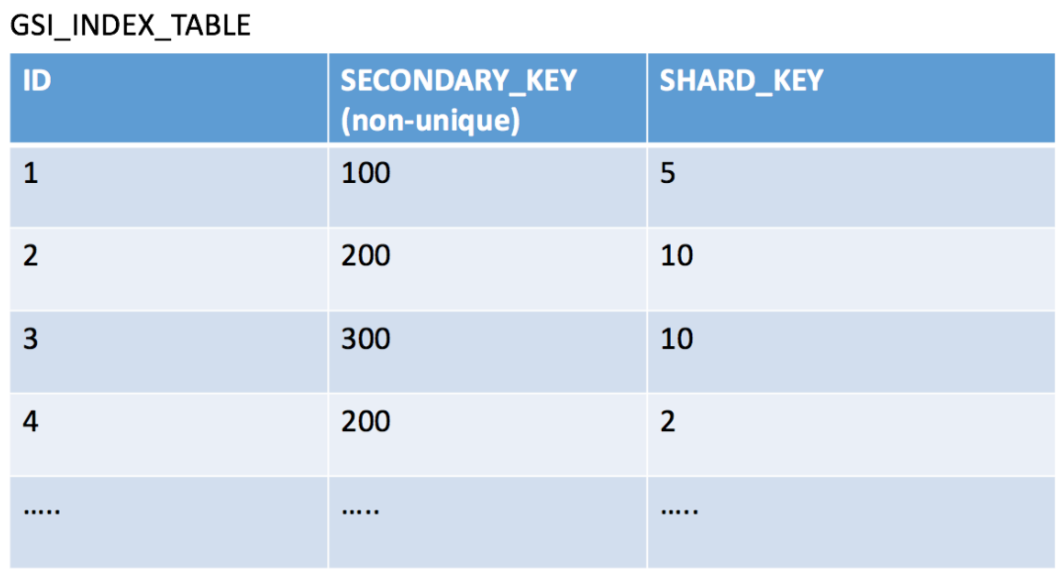
1. All writes must be idempotent so that subsequent retries are successful.

2. Asynchronously fix the tables with inconsistent data.
3. Leave the unreferenced rows dangling and run a reconciliation job periodically.

Read Interface Modifications (Global Secondary Index)

Loads by secondary key (query with no shard key) can't be supported in a sharded environment. A system can have these kind of queries to support various business use cases. For instance, load-by-email-address scenario where a customer agent tries to load customer information based on the customer's email address. If customer-id is the shard key, this load is a secondary key based load.

To address such queries we use a mapping table referred as global secondary index (GSI). GSI provides mapping from secondary key to shard key. Every secondary key based load first obtains the shard key from the mapping table to perform the actual load. Since the lookup is standard and repeated for each entity, investing in a GSI framework to manage these indexes is a good approach. In rest of this section, we discuss the design principles and failure handling mechanisms associated with GSI framework.



GSI_INDEX_TABLE

ID	SECONDARY_KEY (non-unique)	SHARD_KEY
1	100	5
2	200	10
3	300	10
4	200	2
.....	.....	.....

Figure 5. GSI table showing a mapping of secondary key to shard

key

Design Principles

1. Application should be completely agnostic of the secondary index tables.
2. Addition of new secondary indexes or updates to the existing secondary indexes should not require changes in the application.
3. Failure handling and consistency mechanism for GSI should be isolated to the GSI framework level.
4. GSI framework should contain a metadata catalog/table defining important information related to the GSI. Specifically, it should contain primary table name, GSI table name, secondary key, shard key

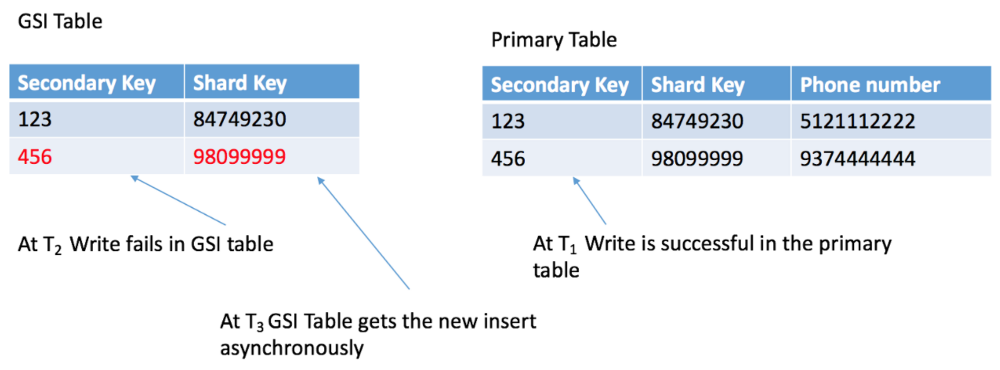
Handling Failures and Maintaining Consistency between Primary and GSI Table

It is critical that the data between primary and GSI table is consistent. Data inconsistency can manifest if a record is successfully inserted in only one of the two tables. In such scenarios, order of the insert operations determine how failures are handled.

Option 1- A record is inserted in the primary table first and insertion in GSI table fails

1. Assuming failure to insert in GSI table doesn't cause failure in the overall operation, this approach favors availability over consistency.
2. Such failures could be fixed asynchronously in the GSI table. If the update to index table is delayed more than the cache expiration time, we can get data inconsistency related issues (figure 6).
3. This implementation is relatively simple to implement and there is no hard dependency on the secondary table in the steady state.
4. This method could also be used if updates are required in the GSI

table.


GSI Table
Primary Table
At T_2 Write fails in GSI table
At T_1 Write is successful in the primary table
At T_3 GSI Table gets the new insert asynchronously

GSI Table

Secondary Key	Shard Key
123	84749230
456	98099999

Primary Table

Secondary Key	Shard Key	Phone number
123	84749230	5121112222
456	98099999	9374444444

Figure 6. Lag (t3-t2) < Cache expiration time

Option 2 - A record is inserted in the GSI table first and write to the primary table fails.

1. If the write to primary table fails, it leaves a dangling row in the GSI table. As long as writes to the mapping table are idempotent, this method offers consistency over availability.
2. This technique works for insert operations only. If GSI tables need updates this technique doesn't work.
3. There needs to be a reconcilation job that periodically cleans the dangling rows.

In order to support transaction integrity, we chose to write to mapping table before writing to the primary table.

Managing Performance in the Presence of Multiple Shard Keys

A secondary key could be associated with multiple shard keys. For example, a home phone number could be associated with all family members yielding many shard keys (customer id) for the same secondary key. In such cases, it is important to understand the use case. If the use case is for batch processing and the number of expected shard keys is huge, redirecting queries to a second tier desharded database is a good option. However, if the query is meant for live use cases where latency impacts the end user, it is best to parallelize the queries.

Challenges Associated with Data Model Transformation

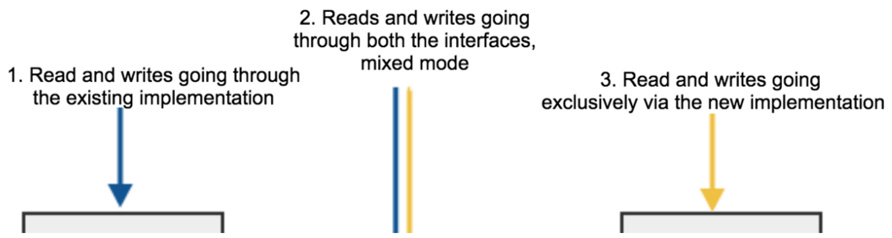
Maintenance of two versions of the tables

The biggest challenge in transforming a high-volume data model is that live migrations can be extremely complex. In addition to that, if there are strong dependencies with downstream customers (reporting teams, business analytics teams, etc.), we need to ensure both the versions of tables are maintained until the teams are able to consume the newer version. In our implementation, we used dual-write strategy where writes are made to two different versions of the table in the application layer. Failures are managed asynchronously.

Complex Transformation Challenges

Denormalization of tables that take heavy traffic is a complex transformation which requires careful designing and execution. In the next section, we describe some of the challenges and the ways we have successfully addressed them in our implementation.

Implementation with mixed mode in consideration - Our feature rollouts are throttled to manage risks. This translates in some servers running older and others running newer version of the software referred to as mixed mode scenario. Supporting this scenario adds some design complexity, especially when a feature is delivering changes related to denormalization of the tables or any other form of the data model change.


1. Read and writes going through the existing implementation
2. Reads and writes going through both the interfaces, mixed mode
3. Read and writes going exclusively via the new implementation

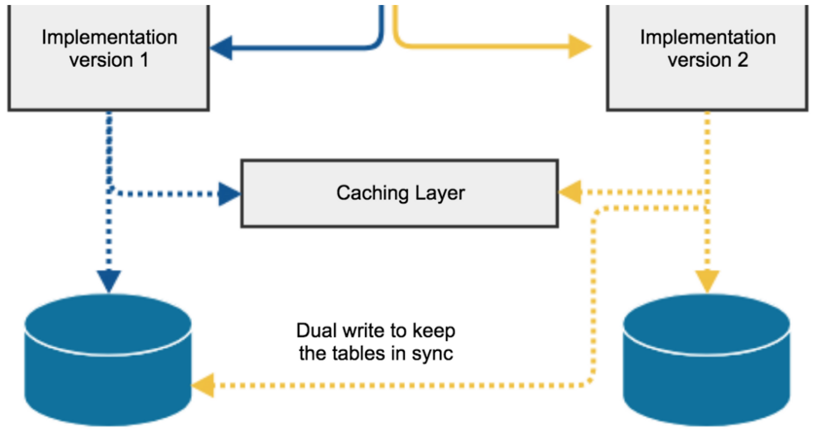


Figure 7. Implications during mixed mode scenario and possible rollback

Failures due to different cache objects - Consider a scenario where a new feature is rolled out. The feature introduces reading and writing from new tables, while the existing tables are still taking traffic (step 2 in figure 4). Now imagine that an insert comes via old code path followed by a read that comes via new code path. The read in this scenario would fail, if the cache object for the old and the new model is different and data is not propagated in the replica database. We address such scenarios by introducing cache objects for both the versions before rolling out the new feature. Assuming, denormalization doesn't involve discontinuation of columns, deriving a cache object from one version to another is simple. Steps involved in releasing a denormalization feature:

1. Introduce cache object mapping from the old to the new version. Each write operation populates both kinds of objects in the cache.
2. The new version of code also writes both versions of the cache objects. The new version continues to write to the old version of the tables until the feature stabilizes. Since, data model changes are done prior to data partitioning into multiple shards, dual writes are done in the same transaction. Dual writes to cache and old tables are discontinued after few weeks.
3. There must be a gap between step 1 and 2 to ensure new cache

object is available for the new version. This depends upon the life span of the cache object.

Monitoring and Validation

Changes of this magnitude require diligence both during data migration and during feature rollouts. There are multiple dimensions like latency, correctness and consistency that are monitored. Here are some strategies we have in place to ensure correctness,

1. *Data consistency checks* -for data migration projects perform multiple levels of validations. We use ora-hash validation, custom scripts and validation for each copy of the data. To expose code related issues, these checks are repeated after feature is live for a day or week.
2. *Logging-* Data model changes can introduce subtle bugs in the code. In our experience, monitoring pattern deviation is very useful. Consider a scenario where a load is performed before inserting a new record in a table. This query would return no data in such scenarios (valid). Capturing this metric can help to identify situations where there are issues due to new changes in the data model. Any significant deviation in the frequency is indicative of a bug.
3. *Special tools* - the process of uncovering bugs should be automated as far as possible. For instance, we developed a tool that would compare different metrics of an application between two time instances. This tool is generic and works for any application. Post ramp, this tool is used for both the application where the changes are introduced as well as for important clients of the application. Similarly for data consistency validation, we created a daemon that verifies if data between two data stores is consistent.

Summary

This article is a detailed technical exercise in exploring how we sharded a highly business critical application DB while keeping the existing business running. As an analogy, this is akin to changing the engines on a flying plane. We believe that these lessons are applicable for any organization undergoing platform scaling and will allow them to anticipate some common problems that they will encounter in their journey. We welcome any feedback.

References

1. Eventually consistent — revisited
2. Scaling large business critical systems
3. How sharding works
4. Beginner’s guide to acid and database transactions