Challenges of Sharding MySQL

Standard / by Dave Anselmi (https://www.clustrix.com/author/dave-anselmi/) / June 26, 2017 / No Comments (https://www.clustrix.com/bettersql/challenges-sharding-mysql/#respond)

MySQL databases are sharded for the best of reasons. At some point the MySQL workload needs to scale, and scaling RDBMSs is hard (Clustrix's unofficial motto!). Adding read scale to RDBMSs is straightforward via replication. But adding write scale? That's the real trick, and herein lies some of the challenges of sharding MySQL. After scaling up your MySQL instance to the largest instance available, and adding several read slaves, what's the **next step (https://www.clustrix.com/event/tech-talk-series-part-1/)**? Multi-master solutions can add additional write scale, but only for separate applications; each application must write to a different master to get that scale. If you have a single MySQL application needing write scale, i.e., ability to fan out writes to multiple MySQL servers simultaneously, MySQL DBAs often start to investigate sharding.

What is Sharding

Sharding is a scale-out approach in which database tables are partitioned, and each partition is put on a separate RDBMS server. For MySQL, this means each node is its own separate MySQL RDBMS, managing its own separate set of data partitions. This data separation allows the application to distribute

queries across multiple servers simultaneously, creating parallelism and thus increasing the scale of that workload. However, this data and server separation also creates challenges, including sharding key choice, schema design, and application rewrites. *Additional challenges of sharding MySQL include data maintenance, infrastructure maintenance, and business challenges, and will be delved into in future blogs.*

Design Challenges of Sharding MySQL

Before an RDBMS can be sharded, several design decisions must be made. Each of these are critical to both the performance of the sharded array, as well as the flexibility of the implementation going forward. These design decisions include the following:

- Sharding key **(https://blog.sqlauthority.com/2016/08/17/database-sharding-identify-shard-key/)** must be chosen
- Schema Changes
- Mapping between sharding key, shards (databases), and physical servers

Choosing a Sharding Key

The sharding key controls how data is distributed across the shards. When sharding a MySQL database, care must be taken when choosing the sharding key, because that decision can later cause system inflexibility. For example referential integrity, the parent/child relationship between tables usually maintained by an RDBMS, won't be automatically maintained if the parent and child rows are on separate databases. This is such a significant challenge that even Google Spanner (which internally shards) avoids the issue by requiring parent/child referential integrity decisions to be made at table **design time (https://www.clustrix.com/bettersql/challenges-googles-cloud-spanner/)**.

There are two main choices for a sharding key:

- **Intelligent sharding keys** help avoid cross-node transactions, but can be more exposed to skew. For instance, if the user table is sharded by user_id, then it's a good idea to put all associated information about that user, including user interactions, user contact points, etc., all on the same shard, avoiding cross-node JOINs. This can work well if a range of users are associated with each other; for instance, if they're all users of the same game (i.e., "all new gaming users go to shard x"). But this can create hotspots, because new users often spike usage.
- **Hashed sharding keys** are automatically distributed across the shards, but can be more exposed to cross-node transactions. For instance, if the sharding key is hash-distributed, then all associated information about the users will also be hash-distributed. This works well to spread out load, especially as new users can spike growth. However, groups of users who interact frequently will require cross-node JOINs, and/or cross-node replication to be in place, creating ongoing latency and/or potentially stale data when there's replication lag.

Another reason choosing the sharding key is critical is because changing an in-place sharding key can be very involved and troublesome. Since sharding key changes can have knock-on effect across application, data location, and transactionality (ACID as well) across nodes, they are usually avoided if at all possible.

Challenges of Schema Changes

That's not to say that schema changes can't be made later. Each MySQL shard can definitely deploy an online schema change, allowing no lost transactions. But the question isn't one of RDBMS support; instead it's that of coordinating all the shards' DDL updates with the application state.

Each of the shards has to successfully complete their online schema change(s), before the application code leveraging the new schema(s) can be enabled. If even one of the shards hasn't completed the change, then data inconsistencies can occur, if not application errors. This kind of coordination across multiple separate MySQL instances is a very involved process, exposing the application to potential errors and downtime. As a result, most administrators of sharded arrays seek to avoid schema changes if at all possible.

For example, at Pinterest, online schema changes were found to be so costly they chose to handle attribute type changes via JSON. Instead of saving new attributes in their own column(s), requiring schema changes to all the shards, Pinterest creates, modifies, and captures new attribute types ad hoc in JSON objects. In other words, the performance tradeoff is beneficial compared to changing the schema, especially across a high number of shards.

Mapping between sharding key, shards (databases), and physical servers

Part of sharding is creating and maintaining the mapping between sharding keys, data partitions, databases, and nodes. This really shouldn't be done in application code, because this mapping will change often, especially in a growing sharded array. Shard splits, shard migrations, instance replacement, and sharding key changes all will change this mapping. Ideally this mapping should be done in a very fast lookup, because potentially this information is needed for each query. Thus this information is often located in an in-memory NoSQL database, such as Redis or Memcache.

Application Changes Required for MySQL Sharding

Sharding usually requires significant application changes as well. Applications moving to a MySQL sharded array now have to handle the following:

- Application query routing to the correct shard
- Cross-node transactions and ACID transactionality

Application Queries Need to Route to Correct the Shard(s)

Since a MySQL application expects a single RDBMS, sharding that single RDBMS requires that the application be able to fan out queries across multiple separate MySQL RDBMSs. Specifically, queries have to be modified to leverage sharding key, and the mapping between that sharding key, shard id, and the server on which that shard currently resides. As described before, the latter information is typically contained in a lookup, requiring additional code to keep up-to-date local version(s) of that mapping data in the application.

Cross-Node Transactions and ACID Transactionality

Having the query access the correct shard is only part of the challenge. Application queries typically need to access lots of data in many different tables, not all of which will be local to the shard. Some of that data will reside in very large tables, which are partitioned and housed on different shards. This affects queries between those large tables, or even between a very large table and itself (what is called a “self-JOIN”). And some of the data resides in smaller tables, which can either be stored locally, or sharded across a smaller subset of the shard array.

Does the application need to access data from multiple shards in a single database transaction? And furthermore, do those transactions need to have **ACID compliance (https://www.clustrix.com/acid-database/)**? If the application needs to support cross-node transactions, then all the JOIN functionality which automatically works in a single-node MySQL JOIN transaction will have to be (re)created in the application itself. The application will have to query one shard, then query a different shard, and then build its own relational logic between the two. If there are uniqueness or parent/child constraints needing to be enforced, those have to be (re)created in the application. And finally, if transactions updating two or more different shards in the same transaction are required, all of that (**Atomicity, Consistency, Isolation, and Durability (https://www.clustrix.com/bettersql/acid-compliance-means-care/)**) will have to be (re)created in the application as well. Basically, “cross-server RDBMS functionality” must be (re)created at the application level, requiring a lot of time and effort, and exposing the application to a lot of risk of data inconsistencies and/or corruption.

Another option is to create an array of cross-node replication between the shards, allowing each shard to have a local copy of data it needs to access. But this is limited to read-only access of that data, and has the potential of being stale due to replication lag. All of the options for implementing or avoiding cross-node transactions have concomitant effects on the workload, and the business rules which can be supported. *Each of these will be explored in a future blog.*

Application Challenges of Sharding MySQL Summary

Sharding MySQL provides scale for MySQL applications, allowing the applications to fan out queries across multiple servers in parallel. But there are challenges of sharding MySQL in order to achieve this scale. By not having a single RDBMS managing ACID transactionality across the shards, this

functionality either has to be avoided (limiting workload and business flexibility), or has to be (re)built at the application level (at high cost and potential data risk).

Tune in to our following blogs when we explore additional challenges of sharding MySQL, including data maintenance, infrastructure and HA considerations, business rule (in)flexibility, and other MySQL sharding options.

Follow Clustrix

(https://twitter.com/Clustrix) **(https://www.linkedin.com/company/clustrix)**
(https://www.facebook.com/Clustrix/)
(https://www.youtube.com/channel/UCvyIIlvRZuP2hhtG_lnup_g)

Visit our **resources (https://www.clustrix.com/resources/)** or **documentation (http://docs.clustrix.com)** page for further reading.

(https://www.facebook.com/Clustrix) (https://twitter.com/Clustrix)
(https://plus.google.com/111948679378130082453) (https://www.linkedin.com/company/clustrix)

Product

Overview (https://www.clustrix.com/summary-of-our-db/)

Case Studies

Hit Labs (https://www.clustrix.com/resources/customer-success-story-hitlabs/)

Recent News

MySQL-compatible cloud database cost comparison (https://www.clustrix.com/bettersql/mysql-compatible-cloud-

Support (https://support.clustrix.com/)

Documentation (http://docs.clustrix.com)

Cloud Database (https://www.clustrix.com/cloud-database/)
Elastic Scale (https://www.clustrix.com/elastic-scale/)
Scale-out Architecture (https://www.clustrix.com/scale-sql/)

Solutions

Ad Tech (https://www.clustrix.com/adtech-database/)
E-commerce (https://www.clustrix.com/ecommerce-database/)
Gaming (https://www.clustrix.com/gaming-database/)
SaaS (https://www.clustrix.com/saas-database/)
Social (https://www.clustrix.com/social-database/)

Match.com (https://www.clustrix.com/resources/customer-success-story-twoo-com/)
Viverae (https://www.clustrix.com/resources/customer-success-story-viverae/)
MakeMyTrip (https://www.clustrix.com/resources/customer-success-story-makemytrip/)

About Us

Management Team (https://www.clustrix.com/management-team/)
Board of Directors & Investors (https://www.clustrix.com/bodinvestors/)
Partners (https://www.clustrix.com/partners/)
Careers (https://www.clustrix.com/careers/)

database-cost-comparison/)

Ad Tech Carousel (https://www.clustrix.com/slideshow/ad-tech-carousel/)

Gaming Carousel (https://www.clustrix.com/bettersql/gaming-carousel/)

SaaS Carousel (https://www.clustrix.com/slideshow/saas-carousel/)

Social carousel (https://www.clustrix.com/slideshow/social-carousel/)

Blog (/bettersql/)
Resources (https://www.clustrix.com/resources/)
Free Trial (https://www.clustrix.com/free-trial/)
Privacy Policy (https://www.clustrix.com/privacy-policy/)

Search …

SEARCH

1.877.806.5357
sales@clustrix.com
(mailto:sales@clustrix.co