Sharding the Hibernate Way

Saturday, July 26, 2008 at 3:04AM

Todd Hoff in Shard, hibernate

Update: *A very nice JavaWorld podcast interview with* Google engineer Max Ross on Hibernate Shards. *Max defines Hibernate Shards (horizontal partitioning), how it works (pretty well), virtual shards (don't ask), what they need to do in the future (query, replication, operational tools), and how it relates to Google AppEngine (not much).*

To scale you are supposed to partition your data. Sounds good, but how do you do it? When you actually sit down to work out all the details it's not that easy. Hibernate Shards to the rescue! Hibernate shards is: an extension to the core Hibernate product that adds facilities for horizontal partitioning. If you know the core Hibernate API you know the shards API. No learning curve at all. Here is what a few members of the core group had to say about the Hibernate Shards open source project. Although there are some limitations, from the sound of it they are doing useful stuff in the right way and it's very much worth looking at, especially if you use Hibernate or some other ORM layer.

Information Sources

1. Google Developer Podcast Episode Six: The Hibernate Shards Open Source Project. This is the document summarized here.
2. Hibernate Shards Project Page
3. Hibernate Shards Dev Discussion Group.
4. Ryan Barrett's Scaling on the Cheap presentation. Many of the lessons from here are in Hibernate Shards.

5. JavaWorld podcast interview: Sharding with Max Ross - Hibernate Shards - Max Ross is the Google engineer who spends his days working on the Google App Engine data store. On the side he works on Hibernate Shards, another scalability-obsessed project that is open source.

What is Hibernate Shards?

1. Shard: splitting up data sets. If data doesn't fit on one machine then split it up into pieces, each piece is called a shard.
2. Sharding: the process of splitting up data. For example, putting employees 1-10,000 on shard1 and employees 10,001-20,000 on shard2.
3. Sharding is used when you have too much data to fit in one single relational database. If your database has a JDBC adapter that means Hibernate can talk to it and if Hibernate can talk to it that means Hibernate Shards can talk to it.
4. Most people don't want to shard because it makes everything complex. But when you have too much data, when you fill your database up, you need another solution, which can be to shard the data across multiple relational databases. The complexity arises because your application has to have the smarts to access multiple databases and that's where Hibernate Shards tries to help.
5. Structure of the data is identical from server to server. The same schema is used across all databases (MySQL, etc).
6. Hibernate was chosen because it's a good ORM tool used internally at Google, but to Google Scale (really really big), sharding needed to be added because Hibernate didn’t support that sort of scale out of the box.
7. The learning curve for a Hibernate user is zero because the Hibernate API is the same. The shard implementation hasn’t violated the API (yet). Sharded versions of Session, Critieria, and

Factory are available so the programmer doesn't need to change code. Query isn't implemented yet because features like aggregation and grouping are very difficult to implement across databases.
8. How does it compare to MySQL's horizontal partitioning? Shards is for situations where you have too much data to fit in a single database. MySQL partitioning may allow you to delay when you need to shard, but it is still a single database and you’ll eventually run into limits.

Schema Design for Shards

1. When sharding you have to consider the general issues of distributed data design for high data volumes. These aren’t Hibernate Shards specific issues, but are general to the problem space.
2. Schema design is the most important of the sharding process and you’ll have to do that up front.
3. You need to pick a dimension, a root level entity, that is easily sharded. Users and customers are common examples.
4. Accept the fact that those entities and all the entities that hang off those entities will be stored in separate physical spaces. Querying across different shards will be difficult. As will management and just about anything else you take for granted.
5. Control over how data are distributed is determined by a pluggable strategies layer.
6. Plan for the future by picking a strategy that will last you a long time. Repartitioning/resharding the data is operationally very difficult. No management tools for this yet.
7. Build simpler models that don't contain as many relationships because you don't have cross shard relationships. Your objects graphs should be contained on one shard as much as possible.
8. Lots of lots of objects pointing to each other may not be a good

candidate for sharding.

9. Because the shards design doesn’t modify Hibernate core, you can design using shards from the start, even though you only have one database. Then when you need to start scaling it will be easier to grow.
10. Existing systems with shardable tables shouldn’t take very long to get up and running.
11. Policy decisions can drive sharding. For example, let's say customers don't want their data intermingling, so each customer would get their own database. In this case the application would shard on the customer as a matter of policy, not simply scaling concerns.

The Sharding Code’s Relationship to Hibernate

1. Hibernate Shards encapsulates knowledge of all the shards. This knowledge is not in the database or the application. It's at the Hibernate persistence layer which provides a unified view of all the databases so the application doesn't have to know.
2. Shards doesn't have full support for Hibernate’s query interface. Hibernate has a criteria or a query interface. Criteria interface is robust, but not good for JPA (Java persistence API), which is query based.
3. Sharding should work across all databases Hibernate works on since shards is a layer on top of Hibernate core beneath the standard Hibernate interfaces. Programmers aren’t aware of it.
4. What they are doing is figuring out how to do standard things like save objects, update, and query objects across multiple databases using standard Hibernate interfaces. If Hibernate can talk to it they can talk to it.
5. A sharded session is used to contain Hibernate’s sessions so

Hibernate capabilities are preserved.

6. Can not manage cross shard foreign relationships (yet). Do have runtime checks to detect when cross shard relations are used accidentally. No foreign key constraint checking and there’s no Hibernate lazy loading. From a programming perspective you can have IDs that reference other objects on other shards, it’s just that Hibernate won’t know about these relationships.
7. Now that the base software is done these more advanced features can be considered. It may take changes in Hibernate core

Pluggable Strategies Determine How Data Are Split Across Shards

1. A Strategy dictates how data are spread across the shards. It’s an interface you need to implement. There are three Strategies:
   * Shard Resolution Strategy - how you will retrieve your objects.
   * Shard Selection Strategy – define where objects are saved to.
   * Access Strategy – once you figure out which shard you are talking to, how do you want to access those shards (serially, 2 at a time, in parallel, etc)?
2. Goal is to have Strategies as flexible as possible so you can decide how your data are sharded.
3. A couple of implementations are provided out of the box:
   * Round Robin - First one goes to the first shard, second to the second shard, and then it loops back.
   * Attribute Based – Look at attributes in the data to determine which shard. You can shard users by country, for example.
4. Configuration is set by creating a prototype configuration for all shards (remember, same schema). Then you specify what's different from shard to shard like URL, user name and password, dialect (MySQL, Postgres, etc). Then they'll create a sharded session factory for Hibernate so developers use standard interfaces.

Some Limitations

1. Full Hibernate HQL is not yet supported (maybe it is now, but I couldn't tell).
2. Distributed queries are handled by applying a standard HQL query to each shard, merging the results, and applying the filters. This all happens in the application server so using very large data sets could be a problem. It's left to the intelligence of the developers to do the right thing to manage performance.
3. No mirroring or data replication. Replication is having common tables, like zip codes, available on all shards.
4. No clean way to manage read only data you want on every shard for performance and referential integrity reasons. Say you have country data. It makes sense to replicate that data on each shard so all queries using that data can stay on the shard.
5. No handling of fail over situations, which is just like Hibernate. You could handle it in your connection pool or some other layer. It's not considered part of the shard/OR mapping layer.
6. There's a need for management tools that work across shards. For example, repartition data on a live system.
7. It's possible to shard across different databases as long as you keep the same schema in the same in each database.
8. The number of shards you can have is somewhat limited because each shard is backed by a connection pool which is a lot of databases connections. And ORDER_BY operations across databases must be done in memory so a lot of memory could be used on large data sets

Related Articles

1. An Unorthodox Approach to Database Design: The Coming of

the Shard.
