Ademir Constantino Filho
(ademirconstantino@gmail.com): é bacharel em Sistemas de Informação e atua com Java desde 2002. Atualmente atua com desenvolvimento, consultoria e como instrutor para treinamentos.

Particionamento de Banco com o Hibernate Shards

Com a disponibilidade de serviços em alta escala nos últimos anos, empresas com grande volume de dados em serviços de grande porte tendem a aplicar conceitos de computação distribuída, mais especificamente bancos de dados distribuídos como solução de distribuição de dados geograficamente. O framework Hibernate Shards permite o desenvolvimento de soluções de particionamento horizontal com Java.

A equipe de projetistas de um sistema de uma rede social tem como objetivo rever o sistema, que hoje centraliza todos os dados de usuários em um único banco de dados e por isso servidores estão sempre operando no vermelho devido ao alto número de transações de leitura e gravação, o que está comprometendo a disponibilidade do serviço para seus usuários.

Problema como este não é novidade em sistemas centralizados com um grande número de acesso, e que ainda com o grande crescimento no número de usuários e alta carga têm tido dificuldade em operar para atender grandes demandas. O fato é que limitações de hardware comprometem a disponibilidade de sistemas e como solução, o particionamento horizontal de dados e frameworks que auxiliam no desenvolvimento e utilizam este conceito, como o Hibernate Shards, começam a surgir e ganhar popularidade no mercado.

Este artigo pretende introduzir os primeiros passos para a programação utilizando o framework Hibernate Shards, em que na Seção 1 "Introdução" é apresentado o conceito de Database sharding e particionamento horizontal de dados, na Seção 2 "Desenvolvendo uma aplicação de registro e login de usuários" a aplicação proposta é apresentada com os devidos passos de configuração e os conceitos que envolvem a programação utilizando o framework e enfim nas considerações finais uma análise de tendências de aplicação no futuro.

Particionamento horizontal e Database sharding

O conceito de particionamento horizontal de dados permite que uma tabela seja dividida entre várias tabelas com o mesmo número de colunas, mas menos linhas. Por exemplo: uma tabela de Usuários pode ser dividida em quatro tabelas, cada uma armazenando dados dos usuários de sua região (considerando Norte, Sul, Leste e Oeste) e estar localizada em diferentes servidores em sua infraestrutura, melhorando o desempenho e facilitando a manutenção.

O conceito de database sharding é um método de particionamento horizontal que tem ganhado popularidade nos últimos anos pelo enorme aumento de transações e tamanho dos bancos de dados corporativos. Esta particularidade é real para muitos provedores de serviços que utilizam o conceito de SaaS (Software as a Service) e sites de redes sociais. A ideia central aqui é imaginar um banco de dados quebrado em cacos (tradução livre para shard: caco), ou seja, em várias partes, ou ainda distribuído. Este conceito foi criado pelos engenheiros da Google e popularizado pela arquitetura BigTable utilizada nos serviços da Google.

Para o entendimento do conceito de database sharding, considere o sistema de uma empresa, que possui uma matriz e três unidades. Se considerarmos o modelo ideal para este sistema, poderemos considerar o modelo onde cada unidade armazena seus dados, compartilhando ainda os dados em comum da matriz conforme a figura 1.

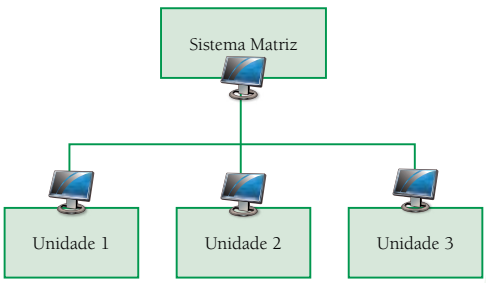

Figura 1. Modelo para um sistema que utiliza o conceito de Database Sharding.

Para implementar soluções em shards em Java, o projeto Hibernate Shards surgiu com o intuito de facilitar o desenvolvimento utilizando os conceitos de database sharding, permitindo ainda que toda a programação possa ser feita utilizando o hibernate, um framework altamente popular na comunidade Java.

Desenvolvendo uma aplicação de registro e login de usuários

Nesta aplicação, o usuário pode efetuar seu cadastro selecionando o banco de dados em que será gravado seus dados, e em seguida, sem distinção de banco de dados o login é efetuado utilizando uma consulta em dois banco de dados.

A aplicação que é usada para exemplificar utiliza JSF, Hibernate Core e Hibernate Shards com annotations e ainda IDE eclipse. O primeiro passo para a criação de nossa aplicação é criar um projeto web no eclipse com faceta Java Server Faces 1.2. Como o foco deste artigo não é o de como criar aplicações em JSF, estes detalhes não são abordados. Após criar a aplicação em JSF é necessário adicionar os jars do hibernate e suas dependências.

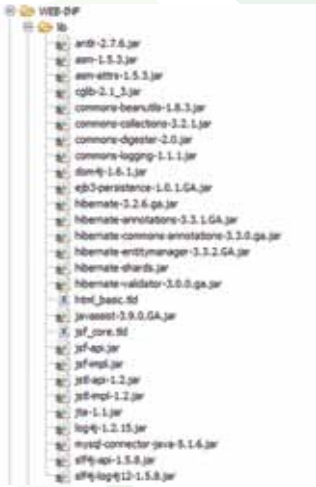

Figura 2. Bibliotecas e dependências do projeto.

Com todos os jars em sua aplicação devemos configurar os arquivos de configuração do hibernate. O hibernate shards utiliza um arquivo de configuração do hibernate para cada banco de dados configurado, podendo ser até mesmo o mesmo host, porém bancos distintos.

Para o aprendizado é utilizado apenas um servidor mysql e dois bancos de dados distintos, porém o primeiro ou segundo banco de dados pode ser alterado para um postgres, Oracle ou até mesmo db2 sem muita dificuldade, apenas alterando o arquivo de configuração referente ao shard que estivermos configurando.

Listagem 1. Código de configuração do shard a.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory name="MySQLSessionFactory">
      <property name="connection.driver_class">com.mysql.jdbc.Driver</
property>
      <property name="connection.url">
            jdbc:mysql://localhost:3306/mjshards_a </property>
      <property name="connection.username">root</property>
      <property name="connection.password">123</property>
      <property name="connection.pool_size">1</property>
      <property name="dialect">
            org.hibernate.dialect.MySQLDialect</property>
      <property name="hibernate.connection.shard_id">0</property>
      <property name="show_sql">true</property>

      <property name="hbm2ddl.auto">none</property>
     <property name="hibernate.shard.enable_cross_shard_relationship_checks">
            true</property>
      <mapping class="br.com.mundoj.mjshards.entity.UsuarioEntity"/>
   </session-factory>
</hibernate-configuration>
```

Listagem 2. Código de configuração do shard b.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory name="MysqlBSessionFactory">
      <property name="connection.driver_class">com.mysql.jdbc.Driver
      </property>
      <property name="connection.url">jdbc:mysql://localhost:3306/mjshards_b
      </property>
      <property name="connection.username">root</property>
      <property name="connection.password">123</property>
      <property name="connection.pool_size">1</property>
      <property name="dialect">org.hibernate.dialect.MySQLDialect
      </property>
      <property name="hibernate.connection.shard_id">1</property>
      <property name="show_sql">true</property>
      <property name="hbm2ddl.auto">none</property>
     <property name="hibernate.shard.enable_cross_shard_relationship_checks">
            true</property>
      <mapping class="br.com.mundoj.mjshards.entity.UsuarioEntity"/>
   </session-factory>
</hibernate-configuration>
```

Os arquivos de configuração são exatamente iguais aos comuns do Hibernate, porém possuem duas propriedades a mais, uma define o id do shard onde são realizados os relacionamentos cruzados. No nó session-factory passa a existir também o atributo name. Isto significa que estamos trabalhando com mais de uma session factory, embora tudo seja controlado pelo framework. O hibernate-shards provê implementações específicas para serem utilizadas quando usados seus recursos. Para isto algumas extensões de classes básicas do Hibernate devem ser utilizadas. Na tabela abaixo é apresentada uma relação das diferenças de classes que são utilizadas quando utilizado o framework Hibernate Shards.

Hibernate Core	Hibernate-shards
rg.hibernate.Session	org.hibernate.shards.session.ShardedSession
org.hibernate.Criteria	org.hibernate.shards.ShardedSessionFactory
org.hibernate.Query	org.hibernate.shards.criteria.ShardedCriteria
org.hibernate.SessionFactory	org.hibernate.shards.query.ShardedQuery

Após criado no mysql os bancos de dados mjshards_a e mjshards_b, devemos criar uma tabela chamada usuário, como é apresentado na Listagem 3.

Listagem 3. Código de criação da tabela Usuário.

```
CREATE  TABLE IF NOT EXISTS  'USUARIO' (
 'ID' INT NOT NULL ,
 'USUARIO' VARCHAR(45) NULL ,
 'SENHA' VARCHAR(45) NULL ,
  PRIMARY KEY ('ID') )
```

É importante compreender que esta tabela deve existir em ambos os bancos de dados. Note que não existe uma constraint para controle de numeração. Isto é importante de ser controlado em nossa aplicação, pois ids não podem ser duplicados em banco de dados quando trabalhamos com o particionamento horizontal. Logo após a criação da tabela em ambos os bancos, deve ser criado também o mapeamento da classe Usuario como apresentado na Listagem 4.

Listagem 4. Código da entidade UsuarioEntity.

```
@Entity
@Table(name = "USUARIO")
public class UsuarioEntity extends AbstractEntity {
   @Id
   @GenericGenerator(name="UsuarioGenerator", strategy="org.hibernate.
shards.id.ShardedUUIDGenerator")
   private Long id;
   @Column(name = "USUARIO")
   private String usuario;
   @Column(name = "SENHA")
   private String senha;
   private transient String idBanco;
   // getters e setters
}
```

Note que o schema não foi configurado para esta entidade. Isto evita que a consulta seja feita em apenas um dos repositórios de dados. Note também que a estratégia de geração de ids foi configurada para ser gerenciada pela aplicação, portanto no momento da gravação de um registro o id será informado. Existem algumas soluções para o gerenciamento de IDS, a saber:

- Native id generation: com a geração de ids nativas do hibernate o banco de dados pode ser configurado para que os ids nunca colidam. Por exemplo, você pode configurar o banco de dados 1 para gerar ids de 0 a 1000 e o banco de dados 2 iniciando de 1000 a 2000 (embora isto seja uma particularidade do banco de dados).
- Geração de id em nível de aplicação: pode ser implementada uma solução para que o gerenciamento de ids seja feito por sua aplicação, dependendo de seus requisitos e regras de negócio. Para isto o hibernate provê um gerador chamado ShardedUUIDGenerator.
- Hilo Generator: com esta solução as chaves são guardadas em uma tabela específica em uma das shards configuradas no hibernate, embora este assunto não seja tratado neste artigo, a documentação possui informações sobre como utilizá-lo.

Após configurarmos nossa entidade no banco de dados e o mapeamento de sua classe, será necessário criar uma classe utilitária para trabalhar com os objetos shards, assim como normalmente é criada uma classe utilitária em aplicações hibernate, temos a seguinte implementação (Listagem 5).

Listagem 5. Classe utilitária do Hibernate.

```
public class ShardedHibernateUtil {

   private static final ShardedSessionFactory sessionFactory;

   static {
      try {
         sessionFactory = createSessionFactory();
      } catch (Throwable ex) {
         throw new ExceptionInInitializerError(ex);
      }
   }

   public static ShardedSessionFactory getSessionFactory() {
      return sessionFactory;
   }

   private static ShardedSessionFactory createSessionFactory() {

      Configuration prototypeConfig = new
            AnnotationConfiguration().
            configure("mysql.shard.configuration.xml");

      ((AnnotationConfiguration)prototypeConfig).
            addAnnotatedClass(br.com.mundoj.mjshards.entity.UsuarioEntity.class);

      List<ShardConfiguration> shardConfigs =
               new ArrayList<ShardConfiguration>();
      shardConfigs.add(buildShardConfig("mysql.shard.configuration.xml"));
      shardConfigs.add(buildShardConfig("mysqlb.shard.configuration.xml"));

      ShardStrategyFactory shardStrategyFactory = buildShardStrategyFactory();

      ShardedConfiguration shardedConfig = new ShardedConfiguration(
            prototypeConfig, shardConfigs, shardStrategyFactory);

      return shardedConfig.buildShardedSessionFactory();
   }

   private static ShardStrategyFactory buildShardStrategyFactory() {
      ShardStrategyFactory shardStrategyFactory = new ShardStrategyFactory() {

         public ShardStrategy newShardStrategy(List<ShardId> shardIds) {
            RoundRobinShardLoadBalancer loadBalancer =
                  new RoundRobinShardLoadBalancer(shardIds);

            ShardSelectionStrategy pss =
                  new MJShardsSelectionStrategy();

            ShardResolutionStrategy prs =
                  new AllShardsShardResolutionStrategy(shardIds);

            ShardAccessStrategy pas = new SequentialShardAccessStrategy();
            return new ShardStrategyImpl(pss, prs, pas);
         }


      };

      return shardStrategyFactory;
   }

   private static ShardConfiguration buildShardConfig(String configFile) {
      AnnotationConfiguration config =
            new AnnotationConfiguration().configure(configFile);
```

Cont. Listagem 5. Classe utilitária do Hibernate.

```
      return new ConfigurationToShardConfigurationAdapter(config);

   }


}
```

O método createSessionFactory acima cria uma configuração para dois bancos de dados utilizando o Hibernate Shards, criando também uma solução de estratégia para a session factory. Uma estratégia define como os dados são armazenados e como eles serão recuperados através de queries. A mais simples das estratégias, ShardAccessStrategy, que é usada para determinar como aplicar operações do banco de dados entre múltiplos shards. A ShardAccessStrategy é consultada sempre que você executar uma query entre suas shards. Duas implementações que utilizam esta interface estão disponíveis.

a) SequentialShardAccessStrategy

SequentialShardAccessStrategy se comporta exatamente do jeito que é implicada por seu nome: queries são executadas em suas shards em sequência. Dependendo do tipo das queries executadas você pode alterar esta implementação porque as queries serão executadas na mesma ordem todas as vezes.

b) ParallelShardAccessStrategy

ParallelShardAccessStrategy também se comporta exatamente como o nome diz: as queries são executadas em suas shards em paralelo. Quando você usar esta implementação você irá precisar prover uma java.util.concurrent.ThreadPoolExecutor que é adaptável para a performance de sua aplicação.

Outra estratégia disponível é a ShardSelectionStrategy usada para determinar em qual shard um novo objeto deve ser criado que será usado neste projeto para definir em qual banco de dados será criado o usuário. Como sua implementação deverá funcionar deve ser totalmente configurada. Algumas soluções podem ser aplicadas utilizando atributos nas tabelas para definir a shard em que se está trabalhando. A estratégia de seleção do projeto está apresentada na Listagem 6.

Listagem 6. Classe contendo a Estratégia de Gravação.

```
public class MJShardsSelectionStrategy implements ShardSelectionStrategy {

   public ShardId selectShardIdForNewObject(Object obj) {
      if (obj instanceof UsuarioEntity) {
         UsuarioEntity usuario = (UsuarioEntity) obj;
         Integer idBanco = Integer.parseInt(usuario.getIdBanco());
         return new ShardId(idBanco);
      }

      return new ShardId(0);
   }

}
```

O seguinte código cria uma estratégia de seleção onde o id do shard a ser gravado é definido pela propriedade UsuarioEntity. idBanco. Neste objeto poderia ser configurada a estratégia para todos os objetos do sistema, bastando incluir condições IF para a instância de classes e as verificações conforme requisitos.

Para a criação de um login, nossa aplicação executará a regra conforme ilustrado na figura 3.

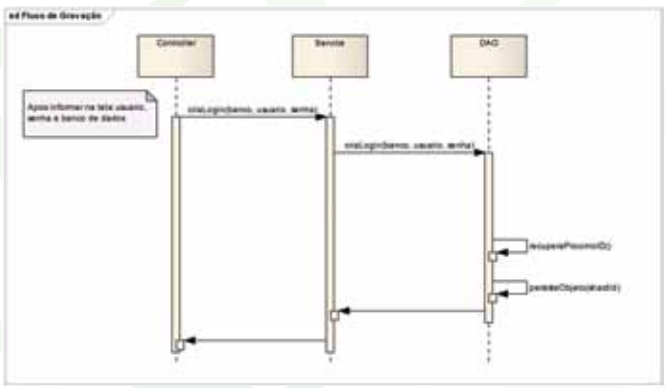

Figura 3. Digrama de sequência ilustrando o fluxo de gravação.

O sistema basicamente disponibiliza uma tela de entrada de dados para que o usuário registre um novo usuário, porém existe uma combo Box indicando qual será o banco de dados onde o dado será gravado. Logo após o registro, caso tudo ocorra com sucesso, é possível validar o usuário através de login. As listagens 7, 8 e 9 apresentam as classes com a implementação.

Listagem 7. Classe de controle da aplicação.

```
public class LoginBean {

   private LoginService loginService;
   private String usuario, senha;
   private String novoUsuario, novaSenha;
   private String banco;

   public LoginBean() {
      loginService = new LoginService();
   }

   /**
    * Método responsável por realizar o login da aplicação
    */
```

cont. Listagem 7. Classe de controle da aplicação.

```
   public String doLogin() throws Exception {

      if(loginService.efetuaLogin(usuario, senha)) {
         return "sucesso";
      } else {
         return "falha";
      }

   }

   public void criarLogin() {

      loginService.criaLogin(banco, novoUsuario, novaSenha);

   }

   // getters e setters

}
```

Listagem 8. Classe de negócio da aplicação.

```
public class LoginService {

   private LoginDAO loginDao = new LoginDAO();

   public boolean efetuaLogin(String usuario, String senha)
   throws Exception {

      UsuarioEntity usuarioVO = (UsuarioEntity) loginDao.doLogin(usuario,
            senha);
      if(usuarioVO != null && usuarioVO.getId() > 0) {
         return true;
      } else {
         return false;
      }
   }

   public void criaLogin(String banco, String usuario, String senha) {
      loginDao.criarLogin(banco, usuario, senha);
   }

   public LoginDAO getLoginDao() {
      return loginDao;
   }

   public void setLoginDao(LoginDAO loginDao) {
      this.loginDao = loginDao;
   }

}
```

Listagem 9. Classe DAO da aplicação.

```
public class LoginDAO extends GenericHibernateDAO<UsuarioEntity> {

   public LoginDAO() {
      super(UsuarioEntity.class);
   }

   /**
    * Método responsável por buscar um login através de usuário e senha
    *
    * @param usuario
    * @param senha
    * @return Um objeto do tipo usuário Entity contendo o usuário logado
    */
   public UsuarioEntity doLogin(String usuario, String senha) {

      ShardedSession s = (ShardedSession)
            ShardedHibernateUtil.getSessionFactory().openSession();
      ShardedCriteria q = (ShardedCriteria) s.createCriteria(
            UsuarioEntity.class).add(Restrictions.eq("usuario", usuario))
            .add(Restrictions.eq("senha", senha));

      UsuarioEntity retornoLogin = (UsuarioEntity) q.uniqueResult();

      s.close();
      return retornoLogin;

   }

   public void criarLogin(String banco, String usuario, String senha) {

      ShardedSession s = (ShardedSession)
            ShardedHibernateUtil.getSessionFactory().openSession();

      Transaction tx = s.beginTransaction();
      UsuarioEntity novoLogin = new UsuarioEntity();
      novoLogin.setUsuario(usuario);
      novoLogin.setSenha(senha);
      novoLogin.setIdBanco(banco);

      novoLogin.setId(0l);

      List<Long> idShard = s.createQuery("select max(usuario.id) from
            UsuarioEntity usuario").list();
      for(Long idMax: idShard) {
         if(idMax != null) {
            if((novoLogin.getId() <= idMax) ) {
               idMax++;
               novoLogin.setId(idMax);
            }
         }
      }

      s.save(novoLogin);
      tx.commit();
      s.close();

   }

}
```

Considerações finais

Com os conceitos de "Software as a Service" e o grande volume de dados, existe uma grande tendência para adoção dos conceitos de database sharding no futuro. Observa-se que em sistemas de redes sociais estes recursos já são utilizados e o framework Hibernate Shards nos permite de forma simples implementar uma solução de particionamento horizontal em Java•

Referências

http://sourceforge.net/projects/hibernate/files/hibernate-shards
http://www.codefutures.com/database-sharding/
http://www.hibernate.org/subprojects/shards.html