

# Inserciones y Consultas Masivas

---

## Inserciones y Consultas Masivas

El proyecto RSI ha venido creciendo y con esto también sus necesidades, en este documento se intentará abordar las diferentes posibilidades para el uso de inserciones masivas o consultas a la base de datos, se presentarán alternativas, y en cada una de ellas algunas de sus ventajas o desventajas.

### Inserciones Masivas

Las inserciones masivas también conocidas por su término en inglés “bulk insert” consisten basicamente en insertar muchos registros en la base de datos de una sola vez. A continuación se presentarán algunas de las opciones que se tienen para mejorar los tiempos en los que ocurren estas inserciones, ya que muchas veces el típico método `saveAll()` que encontramos con los repositorios de *JPA* no llega a ser suficiente para suplir esta necesidad.

#### Hibernate Batch Insert

Para hacer la inserción de multiples registros con el método `saveAll()` Hibernate crea un statement por cada registro y va enviando la instrucción de la inserción de los registros uno por uno, esto es el indicio de la demora cuando se trata de inserciones masivas. Existe un concepto llamado batch insert, y consiste en que estos statements no sean enviados uno por uno, sino que se puedan enviar más a la base de datos, de esta manera las inserciones se harán agrupadas por el número del batch. Por ejemplo si el batch es 30, se enviaran 30 statements de inserciones a la base de datos en lugar de enviarlos uno por uno.



Para que esta característica se active de manera correcta es muy importante que la forma en la que se genera el id de la entidad en cuestión sea de tipo *SEQUENCE*, para las entidades que no tengan este tipo de estrategia se seguirá haciendo la inserción de la manera habitual.

```
1 | @Id
2 | @GeneratedValue(strategy = GenerationType.SEQUENCE)
3 | @Column
4 | private Long id;
```

Configurar la inserción del batch insert en hibernate es muy sencillo, hay que agregar dos propiedades al archivo `application.properties`, que son: `batchsize` y `order_inserts`, la primera para configurar el tamaño del batch, y con la otra propiedad se configura si se quiere que las multiples inserciones se hagan en orden.

```

1 | spring.jpa.properties.hibernate.jdbc.batch_size=30
2 | spring.jpa.properties.hibernate.order_inserts=true

```

Una vez hecho esto, solo es necesario usar el método `saveAll()`, e Hibernate activará la configuración de inserción por batch.

Utilizando este método se encuentran mejoras de hasta un 50%, es decir, si antes para insertar 50.000 registros de la manera clásica se tardaba 100 segundos, ahora se tarda 50 segundos.

## Jdbc Batch Insert

JDBC así como JPA es un API que se comunica con la base de datos, pero a diferencia de JPA, se le deben dar las instrucciones de la base de datos como queries nativas, pero para este método de inserción la estrategia sigue siendo la misma, insertar una gran cantidad de registros, pero dividiendo el número de inserciones y agrupandolas por el tamaño del batch.

A continuación vamos a ver un ejemplo de su forma de uso e implementación.

```

@Override
@Transactional
public void saveAllJdbcBatch(List<ClaseSituacionAdministrativa> cSAEntities, int batchSize) {
    String sql = String.format(
        "INSERT INTO %s (descripcion, senal_grupal, identificador) " +
        "VALUES (?, ?, ?)",
        ClaseSituacionAdministrativa.class.getAnnotation(Table.class).name()
    );
    try (Connection connection = hikariDataSource.getConnection();
        PreparedStatement statement = connection.prepareStatement(sql)) {
        for (ClaseSituacionAdministrativa cSA : cSAEntities) {
            statement.clearParameters();
            statement.setString(1, cSA.getDescripcion());
            statement.setBoolean(2, cSA.getSenalGrupal());
            statement.setString(3, cSA.getIdentificador());
            statement.addBatch();
            if ((counter + 1) % batchSize == 0 || (counter + 1) == cSAEntities.size()) {
                statement.executeBatch();
                statement.clearBatch();
            }
            counter++;
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

```
28 |     }
    | }
```

JDBC utiliza el HikariDataSource que contiene la información de conexión a la base de datos. Primero se escribe el modelo del statement del insert para la inserción masiva, y se van agregando los statement SQL al batch, de acuerdo al tamaño del batch previamente definido, y se va ejecutando el batch una vez este alcanza el tamaño definido.

```
1 | @PostMapping("/bulkInsert/jdbcBatch")
2 | public ResponseEntity<String> saveAllJdbcBatch(@RequestParam int size, @RequestPa
3 |     LocalDateTime startInsert = LocalDateTime.now();
4 |     var data = ClaseSABuilder.builder().setTotal(size).build();
5 |     bulkTestService.saveAllJdbcBatch(data, batchSize);
6 |     var endInsert = LocalDateTime.now();
7 |     return new ResponseEntity<>(size + " data inserted, duration in second "+ Chron
8 | }
```

Usamos una clase Builder para poder generar el tamaño deseado de entidades que vamos a insertar, y calculamos el tiempo que tarda para compararlo con el método saveAll clásico.

Esta es una de las maneras más óptimas de hacer inserciones masivas, las mejoras son bastante considerables. Por ejemplo en las pruebas realizadas para insertar 1000 registros con el método saveAll (sin Hibernate batch) se tarda 60 segundos en promedio, y con el método jdbcBatch, con un size de 30 se tarda 3 segundos.



Este método tiene un problema, y es que al no hacer uso de JPA, no usa las entidades, y por consiguiente no dispara los Listener encargados de hacer la auditoria de base de datos, así que es un método que podría ser usado en tablas que no requieran de auditoría.

## Use connection pool with threading

Otra forma de hacer las inserciones masivas es aprovechando el pool de conexiones y usando hilos o subprocesos que pueden llegar a ser ejecutados al mismo tiempo según el tamaño del pool de conexiones.

A continuación se mostrará un ejemplo del código para dos casos distintos, uno será este método con el saveAll clásico de Hibernate y el otro ejemplo con el jdbcBatch.

```
@Override
@Transactional
public void saveAllThreadsCallable(List<ClaseSituacionAdministrativa> cSAEntities
    ExecutorService executorService = Executors.newFixedThreadPool(hikariDataSour
    List<List<ClaseSituacionAdministrativa>> listOfBookSub = this.createSubList(c
    List<Callable<Void>> callables = listOfBookSub.stream().map(sublist ->
```

```

/
8         (Callable<Void>) () -> {
9             saveAllHibernate(sublist);
10            return null;
11        }).collect(Collectors.toList());
12    try {
13        executorService.invokeAll(callables);
14    } catch (InterruptedException e) {
15        e.printStackTrace();
16    }
}

```

En este fragmento de código lo que se hace es generar una lista de procesos, basada en el batchSize que se le pase, y a su vez la clase Executors hará uso del tamaño máximo del pool de conexiones que también puede ser configurado desde el application.properties de nuestro proyecto. Esto lo que hará es dividir el tamaño de la data y enviar las inserciones masivas en diferentes hilos o subprocessos que serán gestionados de acuerdo al tamaño del pool de conexiones establecido.

Para el ejemplo anterior se obtiene una mejora notable utilizando el SaveAll óptico, por ejemplo, para insertar 1000 registros se pasa de tardar alrededor de 60 segundos a tardar alrededor de 15 segundos en promedio, esta es una mejora bastante significativa, pero no se dispara la auditoría, un truco podría ser tomar el primer registro de la data y hacer un `.save()` para que se dispare la auditoría clásica y el resto de la data si se guarda con este método.

```

1  @Override
2  @Transactional
3  public void saveAllThreadsCallable(List<ClaseSituacionAdministrativa> cSAEntitie
4      ExecutorService executorService = Executors.newFixedThreadPool(hikariDataSou
5      List<List<ClaseSituacionAdministrativa>> listOfBookSub = this.createSubList(
6      List<Callable<Void>> callables = listOfBookSub.stream().map(sublist ->
7          (Callable<Void>) () -> {
8              saveAllJdbcBatch(sublist, batchSize);
9              return null;
10         }).collect(Collectors.toList());
11    try {
12        executorService.invokeAll(callables);
13    } catch (InterruptedException e) {
14        e.printStackTrace();
15    }
16 }

```



El ejemplo anterior es combinandolo ahora con la inserción por medio de JDBC batch, esta es la combinación más eficiente de acuerdo a las pruebas realizadas, pudiendo insertar 50.000 registros con un tamaño del batch de 5.000 en apenas 5 segundos en promedio, teniendo en cuenta que esta forma tampoco dispara la auditoría clásica, pero si se podría aplicar el mismo truco mencionado anteriormente.

## Consultas masivas, uso de caché y demás


### Almacenamiento en caché en Spring-boot

El almacenamiento en caché es útil cuando se tienen que realizar multiples operaciones que son recurrentes, ya que esa información que no cambia con tanta frecuencia puede ser almacenada en un espacio temporal de nuestra aplicación y de esa manera no hay tanto costo temporal a la hora de realizar dichas operaciones, a continuación se mostrará el ejemplo del uso del almacenamiento en caché o *caching* en Spring boot.

Es necesario activar esto por medio de la anotación **@EnableCaching** en la clase principal de nuestra aplicación y así mismo poner la anotación **@Cacheable** sobre el metodo cuyo resultado se quiere que sea guardado en caché.

```
1 | @Cacheable("isEntityAuditable")
2 | @Transactional(propagation = Propagation.REQUIRES_NEW)
3 | public boolean isEntityAuditable(String name, String schema) {
4 |     return tablaAuditoriaLiveRepository.findByNombreAndEsquemaAndActivaTrue(name,
5 | }
```

El ejemplo mostrado anteriormente es un uso de la anotación **@Cacheable** para hacer una busqueda que es bastante recurrente cuando se aplica la auditoría live, esto logró disminuir los tiempos en que se tardaban las peticiones de una manera drástica, ya que para el caso de una tabla que tenía que ser auditada, como mucho hacía la consulta a la base de datos una sola vez, así se implicaran multiples cambios en tablas de base de datos.

Existen multiples configuraciones que pueden hacerse para el manejo de este almacenamiento en caché dentro de la aplicación, en la documentación de [spring](#) , podemos encontrar ejemplos y maneras de usarlo y así sacarle el máximo potencial, como por ejemplo; el tiempo en que se debe guardar la información en caché, o bajo que parámetros esa información siga estando guardada en caché, entre otras cosas.

Algunas de las anotaciones comunmente usadas son **@CacheEvict** que sirve para remover alguna o todas las entradas de algun elemento guardado en caché. La anotación **@CachePut** sirve para agregar entradas al elemento que está guardado en caché, y tiene varias configuraciones dentro de sus parámetros, incluso para que el guardado en caché se haga bajo ciertas condiciones.

