Bases de datos

Hibernate

Mejora de rendimiento de Hibernate

Diego Juarez Romero

2021-2022

Prueba 1: Rendimiento en tiempo

1.1 Caché de segundo nivel

Después de tener todo configurado como se indicaba en las diapositivas de la asignatura, se ejecuta el main del enunciado donde se obtienen los siguientes resultados:

Resultado 2º bloque con la memoria caché desactivada:

Spent acquiring 1 JDBC connections: **173600** nanoseconds
Spent releasing 1 JDBC connections: **66300** nanoseconds
Spent preparing 51000 JDBC statements: **51279800** nanoseconds
Spent executing 51000 JDBC statements: **2127112700** nanoseconds
Spent executing 1 flushes: **5679500** nanoseconds

```
INFO: Session Metrics {
    173600 nanoseconds spent acquiring 1 JDBC connections;
    66300 nanoseconds spent releasing 1 JDBC connections;
    51279800 nanoseconds spent preparing 51000 JDBC statements;
    2127112700 nanoseconds spent executing 51000 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    5679500 nanoseconds spent executing 1 flushes (flushing a total of 1004 entities and 2003 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Resultado 2º bloque con la memoria caché activada:

Spent acquiring 1 JDBC connections: **160600** nanoseconds
Spent releasing 1 JDBC connections: **83100** nanoseconds
Spent preparing 51000 JDBC statements: **48474900** nanoseconds
Spent executing 51000 JDBC statements: **2114269700** nanoseconds
Spent executing 1 flushes: **5888200** nanoseconds
Spent performing L2C puts: **52859000** nanoseconds
Spent performing L2C misses: **24463200** nanoseconds

```
INFO: Session Metrics {
    160600 nanoseconds spent acquiring 1 JDBC connections;
    83100 nanoseconds spent releasing 1 JDBC connections;
    48474900 nanoseconds spent preparing 51000 JDBC statements;
    2114269700 nanoseconds spent executing 51000 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    52859000 nanoseconds spent performing 51004 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    24463200 nanoseconds spent performing 51000 L2C misses;
    5888200 nanoseconds spent executing 1 flushes (flushing a total of 1004 entities and 2003 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Se pueden apreciar diferencias en tiempo sobre todo a la hora de preparar y ejecutar las operaciones JDBC, obteniendo como resultado que es más lento este proceso con la memoria caché de nivel 2 desactivada. Sin embargo se observa que con la memoria caché de nivel 2 activada, ocupa una gran cantidad de tiempo en escribir y leer, lo que hace que el rendimiento sea peor con la memoria de caché de nivel 2 activada.

Después de comentar la línea 66 (session.evict(info);)

Resultado 2º bloque caché desactivada línea 66 comentada:

```
nov. 13, 2021 9:15:05 P. M. org.hibernate.engine.internal.StatisticalLoggingSessionEventListener end
INFO: Session Metrics {
    69800 nanoseconds spent acquiring 1 JDBC connections;
    88100 nanoseconds spent releasing 1 JDBC connections;
    11335800 nanoseconds spent preparing 2883 JDBC statements;
    166413100 nanoseconds spent executing 2883 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    72743600 nanoseconds spent executing 1 flushes (flushing a total of 2004 entities and 2003 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Resultado 2º bloque caché activada línea 66 comentada:

```
nov. 13, 2021 9:15:27 P. M. org.hibernate.engine.internal.StatisticalLoggingSessionEventListener end
INFO: Session Metrics {
    132800 nanoseconds spent acquiring 1 JDBC connections;
    59100 nanoseconds spent releasing 1 JDBC connections;
    12829400 nanoseconds spent preparing 2878 JDBC statements;
    233898300 nanoseconds spent executing 2878 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    139356100 nanoseconds spent performing 3757 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    7482700 nanoseconds spent performing 2000 L2C misses;
    332002600 nanoseconds spent executing 1 flushes (flushing a total of 2004 entities and 2003 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Explica tus conclusiones a partir de este experimento:

Si no se limpia la memoria caché de primer nivel que es lo que se produce al comentar la línea **(session.evict(info);)** , la ejecución con la memoria de segundo nivel desactivada será más rápida (caso contrario al 1º ejemplo) como se puede ver en las imágenes, ya que si tenemos datos

guardados en nuestra memoria de primer nivel no tendremos que realizar acceso a la base de datos para cargar los objetos.
Puesto que se limpiaba para cada iteración, se puede apreciar un gran cambio de rendimiento.

```
for(int i = 0; i<50; i++) {
    for(int idUser: createdUserIds) {
        InformacionPublica info=session.get(InformacionPublica.class,idUser);
        info.setMostrar_email(Math.random() < 0.5);
        info.setMostrar_nacido(Math.random() < 0.5);
        info.setMostrar_nombre(Math.random() < 0.5);
        session.update(info);
        session.evict(info);
    }
}
```

Además el orden de acceso a cache, comienza por caché de primer nivel, a continuación caché de segundo nivel y por último si no se encuentra ninguna información accedemos a la BD (si llegamos a acceder a la base de datos, se guardará en memoria de primer nivel y segundo nivel para mejorar el rendimiento en futuros usos).

1.2 Batches

Resultados agrupadas en "*batch_size*" = 0:

```
INFO: Session Metrics {
    75400 nanoseconds spent acquiring 1 JDBC connections;
    68100 nanoseconds spent releasing 1 JDBC connections;
    4822000 nanoseconds spent preparing 2000 JDBC statements;
    107018000 nanoseconds spent executing 2000 JDBC statements;
    0 nanoseconds spent executing 0 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    148219600 nanoseconds spent executing 1 flushes (flushing a total of 1000 entities and 0 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Resultados agrupadas en "*batch_size*" = 50:

```
INFO: Session Metrics {
    70300 nanoseconds spent acquiring 1 JDBC connections;
    69100 nanoseconds spent releasing 1 JDBC connections;
    72400 nanoseconds spent preparing 2 JDBC statements;
    0 nanoseconds spent executing 0 JDBC statements;
    19029800 nanoseconds spent executing 40 JDBC batches;
    0 nanoseconds spent performing 0 L2C puts;
    0 nanoseconds spent performing 0 L2C hits;
    0 nanoseconds spent performing 0 L2C misses;
    37054600 nanoseconds spent executing 1 flushes (flushing a total of 1000 entities and 0 collections);
    0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0 collections)
}
```

Al cambiar el valor de batch_size se puede ver una gran mejora en el rendimiento, puesto que pasamos de agrupar 0 operaciones en cada batch a agrupar 50 operaciones distintas, esto hace que aumente la velocidad considerablemente.

También se puede observar que mientras antes preparábamos 2000 JDBC statements con "*batch_size*" = 0 ahora con "*batch_size*" = 50 solo se preparan 2 JDBC statements y se ejecutan 40 batches en total.

Por lo que tendremos que tener en cuenta todas estas herramientas a la hora de trabajar con accesos a base de datos, para mejorar considerablemente el rendimiento de nuestro software, siendo capaz de utilizar la opción más conveniente en función del problema surgido para nuestro proyecto.

Prueba 2

1- Se han lanzado 15 consultas diferentes:

La primera consulta es el select a "Usuario" que busca los usuarios en el intervalo de 10-20 que obtiene 11 usuarios diferentes, 11 consultas a "informacion_publica" de cada uno de los usuarios encontrados y 3 consultas a "perfil" puesto que tenemos 3 registros diferentes de perfil disponibles para cada usuario.

2- Añade un bucle for a tu código que recorra los usuarios recuperados y, dentro de este bucle, otro que recorra todas las conexiones de cada usuario.

Después de añadir estos bucles ahora se ejecutan 26 consultas diferentes, esto se debe a que se ha añadido una consulta de "conexión" nueva para cada usuario distinto dentro del bucle, sumando un total de 11 consultas más.

3- Repite el experimento probando:

- Cargas de tipo *LAZY* y *EAGER*
- Estrategias de tipo *SELECT* y *SUBSELECT*

Se ha utilizado SUBSELECT para conexión LAZY:

```
@OneToMany(mappedBy="Usuario", fetch = FetchType.LAZY)
@Fetch(FetchMode.SUBSELECT)
private Set<Conexion> conexion;
```

Mediante este tipo de fetching para “Conexión” dentro de la clase “Usuario”, el resultado del número de consultas ejecutadas cambia notablemente, ahora no necesitamos hacer 11 consultas (una por cada usuario), como las que se hacían anteriormente, ahora solo se ejecuta una consulta.

4- Finalmente, prueba a realizar una consulta con un *JOIN FETCH*. Comenta los resultados obtenidos y tus conclusiones.

- JOIN FETCH sobre Perfil:

```
session.createQuery("select u from Usuario u LEFT JOIN FETCH u.perfil where u.UsuId BETWEEN 10 AND 20").list();
```

Consultas necesarias después del JOIN FETCH:

```
Hibernate: select usuario0_.id_usuario as id_usuar1_4_0_, perfil1_.id_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informaci
Hibernate: select conexion0_.id_usuario as id_usuar2_0_1_, conexion0_.
```

Como podemos ver se reduce el número de consultas necesarias, ya no aparecen las 3 consultas a “perfil” que se ejecutaban anteriormente.

- JOIN FETCH sobre Conexión:

```
"select u from Usuario u LEFT JOIN FETCH u.conexion where u.UsuId BETWEEN 10 AND 20").list();
```

Consultas necesarias después del JOIN FETCH:

```
Hibernate: select usuario0_.id_usuario as id_usuar1_4_0_, conexion1_.momento_entra
Hibernate: select perfil0_.id_perfil as id_perfi1_2_0_, perfil0_.descripcion as de
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select perfil0_.id_perfil as id_perfi1_2_0_, perfil0_.descripcion as de
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select perfil0_.id_perfil as id_perfi1_2_0_, perfil0_.descripcion as de
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_
```

Sin embargo ahora volvemos a necesitar las consultas a “Perfi”pero desaparece la consulta que hacíamos a “Conexión”. Vamos a combinar ambos JOIN FETCH para reducir lo máximo posible el número de consultas a la base de datos.

- JOIN FETCH sobre Perfil y Conexión:

```
("select u from Usuario u LEFT JOIN FETCH u.conexion LEFT JOIN FETCH u.perfil where u.UsuId BETWEEN 10 AND 20").list();
```

Consultas necesarias después del JOIN FETCH:

```
Hibernate: select usuario0_.id_usuario as id_usuar1_4_0_, conexion1_.momento_entrada
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
Hibernate: select informacio0_.id_usuario as id_usuar4_1_0_, informacio0_.mostrar_ema
```

Como se pueden ver en las pruebas realizadas mediante el uso de consultas optimizadas y diferentes tipos de estrategias como cargas de tipo o select/subselect, podemos reducir considerablemente el número de consultas necesarias para mostrar los datos deseados.

Hemos pasado de necesitar **26** consultas a finalmente **11** consultas, para obtener el mismo resultado.