Hibernate thread-safe idempotent upsert without constraint exception handling?

Asked 5 years, 6 months ago Modified 6 months ago Viewed 10k times

20

I have some code that performs an UPSERT, also known as a Merge. I want to clean-up this code, specifically, I want to move away from exception handling, and reduce overall verbosity and sheer complexity of the code for such a simple operation. The requirement is to insert each item unless it already exists:

```
public void batchInsert(IncomingItem[] items) {
    try(Session session = sessionFactory.openSession()) {
        batchInsert(session, items);
    }
    catch(PersistenceException e) {
        if(e.getCause() instanceof ConstraintViolationException) {
            logger.warn("attempting to recover from constraint violation");
            DateTimeFormatter dbFormat = DateTimeFormatter.ofPattern("yyyy-MM-
dd HH:mm:ss.SSS");
            items = Arrays.stream(items).filter(item -> {
                int n = db.queryForObject("select count(*) from rets where
source = ? and systemid = ? and updtdate = ?::timestamp",
                        Integer.class,
                        item.getSource().name(), item.getSystemID(),
                        dbFormat.format(item.getUpdtDateObj()));
                if(n != 0) {
                    logger.warn("REMOVED DUPLICATE: " +
                            item.getSource() + " " + item.getSystemID() + " " +
item.getUpdtDate());
                    return false;
                }
                else {
                    return true; // keep
                }
            }).toArray(IncomingItem[]::new);
            try(Session session = sessionFactory.openSession()) {
                batchInsert(session, items);
            }
        }
    }
}
```

An initial search of SO is unsatisfactory:

- Hibernate Idempotent Update - conceptually similar but much simpler scenario with no regard for multi-threading or multi-processing.
- Can Hibernate work with MySQL's "ON DUPLICATE KEY UPDATE" syntax? much better, removes the race condition by pushing atomicity to the database using `@SQLInsert` annotation; unfortunately, this solution is too error-prone to use on wider tables, and maintenance-intensive in evolving applications.

- How to mimic upsert behavior using Hibernate? very similar to the above question, with a similar answer
- Hibernate + "ON DUPLICATE KEY" logic same as above, answer mentions `merge()` which is ok when single-threaded
- Bulk insert or update with Hibernate? similar question but the chosen answer is off-the-rails, using stored procedures
- Best way to prevent unique constraint violations with JPA again very naive, single-thread-oriented question and answers

In the question How to do ON DUPLICATE KEY UPDATE in Spring Data JPA? which was marked as a duplicate, I noticed this intriguing comment:

> As others have said, there is still an option for using almost the actual same `SQL` statement. Since I would assume that your JPA rep is mostly the default code, you can use `@Modifying @Query("update...")` annotations. – M. Prokhorov Feb 1 '18 at 18:25

That was a dead-end as I really don't understand the comment, despite it sounding like a clever solution, and mention of "actual same SQL statement".

Another promising approach is this: Hibernate and Spring modify query Before Submitting to DB

ON CONFLICT DO NOTHING / ON DUPLICATE KEY UPDATE

Both of the major open-source databases support a mechanism to push idempotency down to the database. The examples below use the PostgreSQL syntax, but can be easily adapted for MySQL.

By following the ideas in Hibernate and Spring modify query Before Submitting to DB, Hooking into Hibernate's query generation, and How I can configure StatementInspector in Hibernate?, I implemented:

```
import org.hibernate.resource.jdbc.spi.StatementInspector;

@SuppressWarnings("serial")
public class IdempotentInspector implements StatementInspector {

    @Override
    public String inspect(String sql) {
        if(sql.startsWith("insert into rets")) {
            sql += " ON CONFLICT DO NOTHING";
        }
        return sql;
    }

}
```

with property

```
        <prop
key="hibernate.session_factory.statement_inspector">com.myapp.IdempotentInspector<
```

Unfortunately this leads to the following error when a duplicate is encountered:

> Caused by: org.springframework.orm.hibernate5.HibernateOptimisticLockingFailureException: Batch update returned unexpected row count from update [0]; actual row count: 0; expected: 1; nested exception is org.hibernate.StaleStateException: Batch update returned unexpected row count from update [0]; actual row count: 0; expected: 1

Which makes sense, if you think about what's going on under the covers: the `ON CONFLICT DO NOTHING` causes zero rows to be inserted, but one insert is expected.

Is there a solution that enables thread-safe exception-free concurrent idempotent inserts and doesn't require manually defining the entire SQL insert statement to be executed by Hibernate?

For what it's worth, I feel that the approaches that push the dupcheck down to the database are the path to a proper solution.

CLARIFICATION The `IncomingItem` objects consumed by the `batchInsert` method originate from a system where records are immutable. Under this special condition the `ON CONFLICT DO NOTHING` behaves the same as an UPSERT, notwithstanding possible *loss of the Nth update*.

java hibernate jpa

Share
Improve this question
Follow

edited Oct 3, 2019 at 3:59

asked Jun 5, 2019 at 2:50

Alex R
11.8k • 18 • 112 • 198

3 Answers

Sorted by: Highest score (default)

10

+100

2023 Update: Hibernate 6.3 introduced an `upsert()` method in `StatelessSession`!

Original Answer from 2019:

Short answer - Hibernate does not support it out of the box (as confirmed by a Hibernate guru in this blog post). Probably you could make it work to some extent in some scenarios with the mechanisms you already described, but just using native queries directly looks the most straightforward approach to me for this purpose.

Longer answer would be that it would be hard to support it considering all the aspects of Hibernate I guess, e.g.:

- What to do with instances for which duplicates are found, as they are supposed to become managed after persisting? Merge them into persistence context?
- What to do with associations that have already been persisted, which cascade operations to apply on them (persist/merge/something_new; or is it too late at that point to make that decision)?
- Do the databases return enough info from upsert operations to cover all use cases (skipped rows; generated keys for not-skipped in batch insert modes, etc).
- What about `@Audit`-ed entities, are they created or updated, if updated what has changed?
- Or versioning and optimistic locking (by the definition you actually want exception in that case)?

Even if Hibernate supported it in some way, I'm not sure I'd be using that feature if there were too many caveats to watch out and take into consideration.

So, the rule of thumb I follow is:

- For simple scenarios (which are most of the time): persist + retry. Retries in case of specific errors (by exception type or similar) can be globally configured with AOP-like approaches (annotations, custom interceptors and similar) depending on which frameworks you use in your project and it is a good practice anyway especially in distributed environments.
- For complex scenarios and performance intensive operations (especially when it comes to batching, very complex queries and alike): Native queries to maximize utilization of specific database features.

Share

Improve this answer

Follow

edited May 24 at 19:13

Alex R

11.8k • 18 • 112 • 198

answered Jun 9, 2019 at 13:08

Dragan Bozanovic

23.5k • 5 • 46 • 114

Thank you. For a native query approach, do you know of a way to get Hibernate to generate and return to me the query that it would ordinarily use for an `INSERT`? That way I can apply `sql += " ON CONFLICT DO NOTHING"` to it and pass it on to the native query API. – Alex R Jun 11, 2019 at 3:49

@AlexR Creative idea, but I'm not aware of any API to do it easily. – Dragan Bozanovic Jun 11, 2019 at 10:50

1

Note that "idempotent" is not the same as "on conflict ignore". The latter may cause the second write to the database to be ignored, even if it actually should do an *update* when the insert fails.

> Is there a solution that enables thread-safe exception-free concurrent idempotent inserts

I'd say that this is probably not even theoretically possible without specific support by the RDBMS, especially the "concurrent" part. The reason is that data will not become actually written and likely not even "visible" until the transaction is committed. So, what would happen if in transaction A it is determined that the record does not exist and an `INSERT` is done. Even if that `INSERT` would be immediately and atomically visible to other transactions, concurrent transaction B would determine that it should do an `UPDATE`. Now, what if later transaction A encounters a problem causing it to be rolled back? The `INSERTED` data from transaction A disappears, and the `UPDATE` of transaction B won't find any record to update.

That's one reason why the "concurrent" part won't work in general, because not all RDBMSs have support for some kind of atomic `UPSERT` (or "on conflict ignore").

However, it seems you don't mind losing the second write (update) to the same record, because you are talking about idempotency, implying that the potential `UPDATE` would not in fact modify the record's data if it already exists. In this case, "on conflict ignore" is indeed equivalent to idempotency.

One (obvious?) 'solution' would be to use some explicit lock (in the database) for mutual exclusion, i.e. transaction A acquires the lock, does its thing, and then releases it again. Transaction B tries to acquire the lock but will be blocked until transaction A is done. This, however, will reduce or prevent concurrency, especially if you process a lot of records in one transaction. Plus, because the RDBMS is not aware of the relation between a lock and the records it guards, the lock is only advisory and every client will have to employ the same locking scheme.

You say that you'd like to "push idempotency down to the database". If that is not a strict requirement, you may be able to just control concurrency in your Java code; e.g.

by using some concurrency-capable collection where your code atomically checks and inserts an ID of each data item it is about to write to the RDBMS. If the ID is already in the collection, skip the item, else insert into DB.

Share Improve this answer Follow

answered Jun 12, 2019 at 10:41

JimmyB
12.6k ● 2 ● 31 ● 47

I added a clarification stating that the `IncomingItem` objects consumed by the `batchInsert` method originate from a system where records are immutable. Under this special condition the `ON CONFLICT DO NOTHING` behaves the same as an `UPSERT`, notwithstanding possible *loss of the Nth update*. You may want to simplify your answer based on this new assumption. – Alex R Jun 12, 2019 at 14:51

-2

I assume based on your post that source, systemid and updtdate is a unique key. Based on that. I would

- retrieve the list of IncomingItem with one query. (I assume that you don't have 1 million records in this DB)
- compare the unique key with your list and keep the one you want to insert.
- save the items

Some pseudo code:

```
public void batchInsert(IncomingItem[] items) {
    //get all IncomingItem from the DB
    List<IncomingItem> incomingItems = //DB query findAll;
    List<IncomingItem> incomingItemsToSave = new ArrayList<>();
    //check your duplicates!
    for(IncomingItem incomingItem : incomingItems){
        Arrays.stream(items).filter(item -> {
            //compare unique key
            // ...  code here ...
            if(!same unique key){
                incomingItemsToSave.add(item);
            }
        });
    }

    try(Session session = sessionFactory.openSession()) {
        batchInsert(session, incomingItemsToSave);
    }
    catch(PersistenceException e) {

    }
}
```

Share Improve this answer Follow

answered Jun 7, 2019 at 10:24

Mannekenpix
752 ● 7 ● 16

This solution fails in a multi-thread / multi-processing environment where the batchInsert() function is invoked multiple times in parallel to maximize performance. Also, my table has millions of records, invalidating one of your assumptions. – Alex R Jun 7, 2019 at 19:00