**DelayedBatchExecutor: How to Optimize Database Usage in Java Multi-Threaded Applications**

In this article, we take a look at a new version of DelayedBatchExecutor and how it helps to optimize multi-threaded applications running concurrently.

[**[](https://dzone.com/users/3981471/victormpcmun.html)**](https://dzone.com/users/3981471/victormpcmun.html)**by**

[**Victor Porcar**](https://dzone.com/users/3981471/victormpcmun.html)

**·**

**Apr. 22, 20 ·**[**Java Zone**](https://dzone.com/java-jdk-development-tutorials-tools-news)**· Tutorial**

**Like**[**(7)**](https://dzone.com/articles/delayedbatchexecutor-how-to-optimize-database-usag)

**Comment (3)**

**Save**

[**Tweet**](https://dzone.com/articles/delayedbatchexecutor-how-to-optimize-database-usag)

**4.97K Views**

I described in my post [Optimizing Data Repositories Usage in Java Multi-Threaded Applications](https://dzone.com/articles/optimizing-data-repositories-usage-in-java-multith) a simple mechanism called [DelayedBatchExecutor](https://github.com/victormpcmun/delayed-batch-executor) to decrease the number of required queries in Java multi-threaded applications by batching them.

While this mechanism worked well, it required to block the threads for an interval of time, which was not optimal in some cases. Now, I have just released a new version of DelayedBatchExecutor in the central repository that includes non-blocking behavior in two ways:

* using [Future](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html)
* using [Reactor](https://projectreactor.io/)[Mono](https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html)(Reactive Programming)

**What is DelayedBatchExecutor?**

As I explained in my previous post, there are several scenarios in which concurrent threads execute many times the same query to a database at almost the same time (each query with a  different parameter).

For example, a REST endpoint serving tens or hundreds of requests per second in which each one requires to retrieve an entity from the database by a different Id.

In a similar way, another typical scenario is a message listener that consumes a large number of messages per second and requires to execute a query by a different Id to process each one.

In all these cases, the database executes many times queries like the following in  a short interval of time (milliseconds):

SQL



1

SELECT \* FROM TABLE WHERE ID   = <id1>

2

SELECT \* FROM TABLE WHERE ID   = <id2>

3

...

4

SELECT \* FROM TABLE WHERE ID   = <idn>

DelayedBatchExecutor is a component that allows easily to *convert* these n queries of 1 parameter into just one single query with n parameters, like this one:

SQL



1

SELECT \* FROM TABLE WHERE ID IN (<id1>, <id2>, ..., <idn>)

The advantages of executing one query with n parameters instead of n queries of 1 parameter are the following:

* **Optimization of the database server**: you would be surprised how well databases optimize queries for *n* parameters, especially the complex ones. The larger is the *n,* the better CPU usage optimization is achieved. Although the actual optimization depends on many factors (table sizes, query complexity,...),  my tests point to a reduction up to 30% of CPU Usage on the Database Server (Oracle) for n>50 per second, and up to 50% for n>100 per second. I suggest you pick a table from your schema and analyze the execution times and CPU usage in both scenarios.
* **The usage of network resources is reduced dramatically**: The number of round-trips to the database is 1 instead of *n*.
* **The usage of connections from the connection pool is reduced**: There are more available connections overall, which means less waiting time for a connection on peak times.

In short, it is much more efficient executing one query of *n* parameters than *n* queries of one parameter, which means that the system as a whole requires less resources.

**How DelayedBatchExecutor Works**

It basically works by creating *time windows* of milliseconds where the parameters of the queries executed during the *time window* are collected in a list. As soon as the *time window* finishes, the list is passed (via callback) to a method that executes one single query with all the parameters in the list and returns another list with the results. Each thread receives its corresponding result from the result list according to one of the following policies as explained below: blocking, non-blocking (Future), and non-blocking (Mono of Reactor framework).

All these actions are performed by the DelayedBatchExecutor transparently behind the scenes.

A DelayedBatchExecutor is defined by three parameters:

* TimeWindow : defined as java.time.Duration
* Max Size : it is the max number of items to be collected in the list
* BatchCallback : it receives the parameters list to perform a single query and must return a list with the corresponding results.
  + It can be implemented as a method reference or lambda expression.
  + It is invoked automatically as soon as the  TimeWindow is finished OR the collection list is full.
  + The returned list must have a correspondence in elements with the parameters list, this means that the value of position 0 of the returned list must be the one corresponding to parameter in position 0 of the param list and so on...
  + By default, duplicated parameters (by [hashCode](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#hashCode--) and [equals](https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-)) are removed from the parameters list automatically. This is optimal in most cases although there is a way for including duplicates (See Advanced Features).

**DelayedBatchExecutor Example**

First, import the dependency in our project:

**Maven**

XML



1

<dependency>

2

 <groupId>com.github.victormpcmun</groupId>

3

 <artifactId>delayed-batch-executor</artifactId>

4

 <version>3.1</version>

5

</dependency>

**Gradle**

Groovy



1

implementation 'com.github.victormpcmun:delayed-batch-executor:3.1'

Second, declare a  DelayedBatchExecutor to receive an integer value as parameter and return a String. In this example, the time window will be 50 milliseconds and  max size will be 100 elements.

It can be declared in two ways:

* Having the batchCallBack defined as method reference:

Java



1

DelayedBatchExecutor2<String,Integer> dbe =

2

 DelayedBatchExecutor2.create(Duration.ofMillis(50), 100, this::myBatchCallBack);

3

4

...

5

6

List<String> myBatchCallBack(List<Integer> listOfIntegers) {

7

   // execute query:

8

   // SELECT \* FROM TABKE WHERE ID IN (listOfIntegers.get(0), ..., listOfIntegers.get(n));

9

// use your favourite API: JDBC, JPA, Hibernate,...

10

   List<String>  resultList = ...

11

   return resultList;

12

}

* Having the  batchCallBack  as a lambda expression:

Java



1

DelayedBatchExecutor2<String,Integer> dbe = DelayedBatchExecutor2.create(Duration.ofMillis(50), 100, listOfIntegers->

2

{

3

   // execute query:

4

   // SELECT \* FROM TABKE WHERE ID IN (listOfIntegers.get(0), ..., listOfIntegers.get(n));

5

// use your favourite API: JDBC, JPA, Hibernate,...

6

 List<String>  resultList = ...

7

 return resultList;

8

});

NOTE: The instance dbe must be accessible from the code being executed by the threads (it is typically declared as instance variable of a singleton DAO).

Third, use DelayedBatchExecutor from the code executed in each thread:

Java



1

// this code is executed in one of the multiple threads

2

int param=...;

3

...

4

String result=dbe.execute(param); //all threads executing this line during the window time (50ms) will have

5

                                 // their parameters collected in the list to be passed to the callback

6

                                 // and from the returned list of the callback

7

                                 // this thread will receive its corresponding result

And that's it.

NOTE:

* In the example above, the thread is stopped when the  execute(...)  method is executed until the result is available (blocking behaviour). This is one of the three execution policies explained below.
* This example shows a DelayedBatchExecutor for one argument of type Integer and return type of String, hence DelayedBatchExecutor2<String,Integer>. For a DelayedBatchExecutor for two arguments (say Integer and Date) and a returning type of String, the definition would be DelayedBatchExecutor3<String,Integer,Date>  and so on.

**Execution Policies**

There are three policies to use a DelayedBatchExecutor from the code being executed from the threads

**Blocking**

The thread is blocked until the result is available, it is implemented by using the method execute(...).

Java



1

int param = ...

2

...

3

String result = dbe.execute(param); // this thread will be blocked until the result is available

4

// compute with result

The following diagram depicts how blocking policy works:

**Non-blocking (java.util.concurrent.Future)**

The thread is not blocked, it is implemented by using the method executeAsFuture(...)

Java



1

int param = ...

2

...

3

Future<String> resultFuture = dbe.executeAsFuture(param); // the thread will not be blocked

4

// compute something else

5

String result = resultFuture.get();  // Blocks the thread until the result is available (if necessary)

6

// compute with result

The following diagram depicts how Future policy works:

**Non-blocking (Reactor Mono):**

The thread is not blocked, it is implemented by using the method executeAsMono(...)

Java



1

int param =...

2

...

3

reactor.core.publisher.Mono<String> resultMono = dbe.executeAsMono(param); // the thread will not be blocked

4

// compute something else

5

resultMono.subscribe(stringResult -> {

6

// compute with stringResult

7

});

The following diagram depicts how Reactor Mono policy works:

**Advanced Features**

There are three parameters of a DelayedBatchExecutor that must be known to get the most of it:

* ExecutorService: The callback method is actually executed in a parallel thread, which is provided by a java.util.concurrent.ExecutorService. By default this Executor is Executors.newFixedThreadPool(4).
* bufferQueueSize: It is the max size of the internal buffer, by default its value is 8192.
* removeDuplicates: A Boolean flag. If it is false, then DelayedBatchExecutor won't removed all duplicated parameters from the parameters list before invoking the  batchCallback . By default its value is true.

These parameters can be set at the declaration time by using the following constructor:

Java



1

...

2

int maxSize=20;

3

ExecutorService executorService= Executors.newFixedThreadPool(10);

4

int bufferQueueSize= 16384;

5

boolean removeDuplicates = false;

6

7

DelayedBatchExecutor2<String,Integer> dbe = DelayedBatchExecutor2.create(

8

   Duration.ofMillis(200),

9

   maxSize,

10

   executorService,

11

   bufferQueueSize,

12

   removeDuplicates,

13

   this::myBatchCallBack);