Prepared Statements

Under the hood, whenever you ask `next.jdbc` to execute some SQL (via `plan`, `execute!`, `execute-one!` or the "friendly" SQL functions) it calls `prepare` to create a `java.sql.PreparedStatement`, adds in the parameters you provide, and then calls `.execute` on it. Then it attempts to get a `ResultSet` from that and either return it or process it. If you asked for generated keys to be returned, that `ResultSet` will contain those generated keys if your database supports it, otherwise it will be whatever the `.execute` function produces. If no `ResultSet` is available at all, `next.jdbc` will ask for the count of updated rows and return that as if it were a result set.

> Note: Some databases do not support all SQL operations via `PreparedStatement`, in which case you may need to create a `java.sql.Statement` instead, via `next.jdbc.prepare/statement`, and pass that into `plan`, `execute!`, or `execute-one!`, along with the SQL you wish to execute. Note that such statement execution may not have parameters. See the Prepared Statement Caveat in Getting Started for an example.

If you have a SQL operation that you intend to run multiple times on the same `java.sql.Connection`, it may be worth creating the prepared statement yourself and reusing it. `next.jdbc/prepare` accepts a connection and a vector of SQL and optional parameters and returns a `java.sql.PreparedStatement` which can be passed to `plan`, `execute!`, or `execute-one!` as the first argument. It is your responsibility to close the prepared statement after it has been used.

If you need to pass an option map to `plan`, `execute!`, or `execute-one!` when passing a statement or

```
  (with-open [ps (jdbc/prepare con [ ...  ...])]
    (jdbc/execute-one! ps nil {...})))
  (with-open [stmt (jdbc/statement con)]
    (jdbc/execute-one! stmt nil {...})))
```

You can provide the parameters in the `prepare` call or you can provide them via a call to `set-parameters` (discussed in more detail below).

```
;; assuming require next.jdbc.prepare :as p
(with-open [con (jdbc/get-connection ds)
            ps  (jdbc/prepare con ["..."])]
  (jdbc/execute-one! (p/set-parameters ps [...])))
```

Prepared Statement Parameters

If parameters are provided in the vector along with the SQL statement, in the call to `prepare`, then `set-parameter` is behind the scenes called for each of them. This is part of the `SettableParameter` protocol:

- `(set-parameter v ps i)` -- by default this calls `(.setObject ps i v)` (for `nil` and `Object`)

This can be extended to any Clojure data type, to provide a customized way to add specific types of values as parameters to any `PreparedStatement`. For example, to have all `java.time.Instant`, `java.time.LocalDate` and `java.time.LocalDateTime` objects converted to `java.sql.Timestamp` automatically:

```
(extend-protocol p/SettableParameter
```

```
  (set-parameter [^java.time.LocalDate v ^PreparedStatement ps ^long i]
    (.setTimestamp ps i (java.sql.Timestamp/valueOf (.atStartOfDay v))))
  java.time.LocalDateTime
  (set-parameter [^java.time.LocalDateTime v ^PreparedStatement ps ^long i]
    (.setTimestamp ps i (java.sql.Timestamp/valueOf v))))
```

> Note: those conversions can also be enabled by requiring the `next.jdbc.date-time` namespace.

You can also extend this protocol via metadata so you can do it on a per-object basis if you need:

```
(with-meta obj {'next.jdbc.prepare/set-parameter (fn [v ps i]...)})
```

The `next.jdbc.types` namespace provides functions to wrap values with per-object implementations of `set-parameter` for every standard `java.sql.Types` value. Each is named `as-xxx` corresponding to `java.sql.Types/XXX`.

The converse, converting database-specific types to Clojure values is handled by the `ReadableColumn` protocol, discussed in the previous section (Result Set Builders).

As noted above, `next.jdbc.prepare/set-parameters` is available for you to call on any existing `PreparedStatement` to set or update the parameters that will be used when the statement is executed:

- `(set-parameters ps params)` -- loops over a sequence of parameter values and calls `set-parameter` for each one, as above.

If you need more specialized parameter handling than the protocol can provide, then you can create prepared statements explicitly, instead of letting `next.jdbc` do it for you, and then calling your own variant

By default, `next.jdbc` assumes that you are providing a single set of parameter values and then executing the prepared statement. If you want to run a single prepared statement with multiple groups of parameters, you might want to take advantage of the increased performance that may come from using JDBC's batching machinery.

You could do this manually:

```
;; assuming require next.jdbc.prepare :as p
(with-open [con (jdbc/get-connection ds)
            ps  (jdbc/prepare con ["insert into status (id,name) values (?,?)"])]
  (p/set-parameters ps [1 "Approved"])
  (.addBatch ps)
  (p/set-parameters ps [2 "Rejected"])
  (.addBatch ps)
  (p/set-parameters ps [3 "New"])
  (.addBatch ps)
  (.executeBatch ps)) ; returns int[]
```

Here we set parameters and add them in batches to the prepared statement, then we execute the prepared statement in batch mode. You could also do the above like this, assuming you have those groups of parameters in a sequence:

```
(with-open [con (jdbc/get-connection ds)
            ps  (jdbc/prepare con ["insert into status (id,name) values (?,?)"])]
  (run! #(.addBatch (p/set-parameters ps %))
        [[1 "Approved"] [2 "Rejected"] [3 "New"]])
```

provided in `next.jdbc` to automate the execution of batched parameters:

```
(with-open [con (jdbc/get-connection ds)
            ps  (jdbc/prepare con ["insert into status (id,name) values (?,?)"])]
  (jdbc/execute-batch! ps [[1 "Approved"] [2 "Rejected"] [3 "New"]]))
;; or:
(jdbc/execute-batch! ds
                     "insert into status (id,name) values (?,?)"
                     [[1 "Approved"] [2 "Rejected"] [3 "New"]]
                     ;; options hash map required here to disambiguate
                     ;; this call from the 2- & 3-arity calls
                     {})
```

By default, this adds all the parameter groups and executes one batched command. It returns a (Clojure) vector of update counts (rather than `int[]`). If you provide an options hash map, you can specify a `:batch-size` and the parameter groups will be partitioned and executed as multiple batched commands. This is intended to allow very large sequences of parameter groups to be executed without running into limitations that may apply to a single batched command. If you expect the update counts to be very large (more than `Integer/MAX_VALUE`), you can specify `:large true` so that `.executeLargeBatch` is called instead of `.executeBatch`.

> Note: not all databases support `.executeLargeBatch`.

If you want to get the generated keys from an `insert` done via `execute-batch!`, you need a couple of extras, compared to the above:

```
(with-open [con (jdbc/get-connection ds)
```

```
  ;; vector of datafiable result sets (the keys in map are database-specific):
  (jdbc/execute-batch! ps [[1 "Approved"] [2 "Rejected"] [3 "New"]]
                       {:return-generated-keys true}))
;; or:
(jdbc/execute-batch! ds
                     "insert into status (id,name) values (?,?)"
                     [[1 "Approved"] [2 "Rejected"] [3 "New"]]
                     {:return-keys true ; for creation of PreparedStatement
                      :return-generated-keys true}) ; for batch result format
```

This calls `rs/datafiable-result-set` behind the scenes so you can also pass a `:builder-fn` option to `execute-batch!` if you want something other than qualified as-is hash maps.

> Note: not all databases support calling `.getGeneratedKeys` here (everything I test against seems to, except MS SQL Server and SQLite). Some databases will only return one generated key per batch, rather than a generated key for every row inserted. You may need to add `RETURNING *` to your `INSERT` statements instead.

Caveats

There are several caveats around using batched parameters. Some JDBC drivers need a "hint" in order to perform the batch operation as a single command for the database. In particular, PostgreSQL requires the `:reWriteBatchedInserts true` option and MySQL requires `:rewriteBatchedStatements true` (both non-standard JDBC options, of course!). These should be provided as part of the db-spec hash map when the datasource is created.

In addition, if the batch operation fails for a group of parameters, it is database-specific whether the remaining groups of parameters are used, i.e., whether the operation is performed for any further groups of

`.getLargeUpdateCounts`) on the exception may return an array containing a mix of update counts and error values (a Java `int[]` or `long[]`). Some databases don't always return an update count but instead a value indicating the number of rows is not known (but sometimes you can still get the update counts).

Finally, some database drivers don't do batched operations at all -- they accept `.executeBatch` but they run the operation as separate commands for the database rather than a single batched command. Some database drivers do not support `.getGeneratedKeys` (e.g., MS SQL Server and SQLite) so you cannot use `:return-generated-keys` and you need to use `RETURNING *` in your `INSERT` statements instead.

❮ Result Set Builders Transactions ❯