Friendly SQL Functions

In Getting Started, we used `execute!` and `execute-one!` for all our SQL operations, except when we were reducing a result set. These functions (and `plan`) all expect a "connectable" and a vector containing a SQL string followed by any parameter values required.

A "connectable" can be a `javax.sql.DataSource`, a `java.sql.Connection`, or something that can produce a datasource (when `get-datasource` is called on it). It can also be a `java.sql.PreparedStatement` but we'll cover that a bit later...

Because string-building isn't always much fun, `next.jdbc.sql` also provides some "friendly" functions for basic CRUD operations:

- `insert!` and `insert-multi!` -- for inserting one or more rows into a table -- "Create",
- `query` -- an alias for `execute!` when using a vector of SQL and parameters -- "Read",
- `update!` -- for updating one or more rows in a table -- "Update",
- `delete!` -- for deleting one or more rows in a table -- "Delete".

as well as these more specific "read" operations:

- `find-by-keys` -- a query on one or more column values, specified as a hash map or `WHERE` clause,
- `get-by-id` -- a query to return a single row, based on a single column value, usually the primary key.

These functions are described in more detail below. They are deliberately simple and intended to cover only

- HoneySQL -- a composable DSL for creating SQL/parameter vectors from Clojure data structures
- seql -- a simplified EQL-inspired query language, built on `next.jdbc` (as of release 0.1.6)
- SQLingvo -- a composable DSL for creating SQL/parameter vectors
- Walkable -- full EQL query language support for creating SQL/parameter vectors

If you prefer to write your SQL separately from your code, take a look at HugSQL -- HugSQL documentation -- which has a `next.jdbc` adapter, as of version 0.5.1. See below for a "quick start" for using HugSQL with `next.jdbc`.

As of 1.3.925, `aggregate-by-keys` exists as a wrapper around `find-by-keys` that accepts the same options as `find-by-keys` and an aggregate SQL expression and it returns a single value (the aggregate). `aggregate-by-keys` accepts the same options as `find-by-keys` except that `:columns` may not be specified (since it is used to add the aggregate to the query).

`insert!`

Given a table name (as a keyword) and a hash map of column names and values, this performs a single row insertion into the database:

```
(sql/insert! ds :address {:name "A. Person" :email "albert@person.org"})
;; equivalent to
(jdbc/execute-one! ds ["INSERT INTO address (name,email) VALUES (?,?)"
                       "A.Person" "albert@person.org"] {:return-keys true})
;; some databases may require this instead
(jdbc/execute-one! ds ["INSERT INTO address (name,email) VALUES (?,?) RETURNING *"
                       "A.Person" "albert@person.org"])
```

If you have multiple rows (hash maps) to insert and they all have the same set of keys, you can use `insert-multi!` instead (see below), which will perform a single multi-row insertion, which will generally be faster.

`insert-multi!`

Given a table name (as a keyword), a vector of column names, and a vector of row value vectors, this performs a single multi-row insertion into the database:

```
(sql/insert-multi! ds :address
  [:name :email]
  [["Stella" "stella@artois.beer"]
   ["Waldo" "waldo@lagunitas.beer"]
   ["Aunt Sally" "sour@lagunitas.beer"]])
;; equivalent to
(jdbc/execute! ds ["INSERT INTO address (name,email) VALUES (?,?), (?,?), (?,?)"
                   "Stella" "stella@artois.beer"
                   "Waldo" "waldo@lagunitas.beer"
                   "Aunt Sally" "sour@lagunitas.beer"] {:return-keys true})
```

All the row vectors must be the same length, and must match the number of columns specified.

Given a table name (as a keyword) and a vector of hash maps, this performs a single multi-row insertion into the database:

```
(sql/insert-multi! ds :address
```

```
(jdbc/execute! ds ["INSERT INTO address (name,email) VALUES (?,?), (?,?), (?,?)"
                   "Stella" "stella@artois.beer"
                   "Waldo" "waldo@lagunitas.beer"
                   "Aunt Sally" "sour@lagunitas.beer"] {:return-keys true})
```

All the hash maps must have the same set of keys, so that the vector of hash maps can be converted to a vector of columns names and a vector of row value vectors, as above, so a single multi-row insertion can be performed.

If you wish to insert multiple hash maps that do not have identical keys, you need to iterate over `insert!` and insert one row at a time, which will generally be much slower.

> Note: both of these expand to a single SQL statement with placeholders for every value being inserted -- for large sets of rows, this may exceed the limits on SQL string size and/or number of parameters for your JDBC driver or your database. Several databases have a limit of 1,000 parameter placeholders. Oracle does not support this form of multi-row insert, requiring a different syntax altogether.

Batch Insertion

As of release 1.2.790, you can specify `:batch true` in the options, which will use `execute-batch!` under the hood, instead of `execute!`, as follows:

```
(sql/insert-multi! ds :address
  [:name :email]
  [["Stella" "stella@artois.beer"]
   ["Waldo" "waldo@lagunitas.beer"]
   ["Aunt Sally" "sour@lagunitas.beer"]])
```

```
                       ["Stella" "stella@artois.beer"]
                       ["Waldo" "waldo@lagunitas.beer"]
                       ["Aunt Sally" "sour@lagunitas.beer"]]
                      {:return-keys true :return-generated-keys true})
;; and
(sql/insert-multi! ds :address
  [:name :email]
  [{:name "Stella", :email "stella@artois.beer"}
   {:name "Waldo", :email "waldo@lagunitas.beer"}
   {:name "Aunt Sally", :email "sour@lagunitas.beer"}]
  {:batch true})
;; equivalent to
(jdbc/execute-batch! ds
                      ["INSERT INTO address (name,email) VALUES (?,?)"
                       ["Stella" "stella@artois.beer"]
                       ["Waldo" "waldo@lagunitas.beer"]
                       ["Aunt Sally" "sour@lagunitas.beer"]]
                      {:return-keys true :return-generated-keys true})
```

> Note: not all databases or drivers support returning generated keys like this -- see **Batched Parameters** for caveats and possible database-specific behaviors. You may need `RETURNING *` in your SQL instead.

`query`

Given a vector of SQL and parameters, execute it:

Note that the single argument form of `execute!`, taking just a `PreparedStatement`, is not supported by `query`.

`update!`

Given a table name (as a keyword), a hash map of columns names and values to set, and either a hash map of column names and values to match on or a vector containing a partial `WHERE` clause and parameters, perform an update operation on the database:

```
(sql/update! ds :address {:name "Somebody New"} {:id 2})
;; equivalent to
(sql/update! ds :address {:name "Somebody New"} ["id = ?" 2])
;; equivalent to
(jdbc/execute-one! ds ["UPDATE address SET name = ? WHERE id = ?"
                       "Somebody New" 2])
```

`delete!`

Given a table name (as a keyword) and either a hash map of column names and values to match on or a vector containing a partial `WHERE` clause and parameters, perform a delete operation on the database:

```
(sql/delete! ds :address {:id 8})
;; equivalent to
(sql/delete! ds :address ["id = ?" 8])
```

find-by-keys

Given a table name (as a keyword) and either a hash map of column names and values to match on or a vector containing a partial `WHERE` clause and parameters, execute a query on the database:

```
(sql/find-by-keys ds :address {:name "Stella" :email "stella@artois.beer"})
;; equivalent to
(sql/find-by-keys ds :address ["name = ? AND email = ?"
                               "Stella" "stella@artois.beer"])
;; equivalent to
(jdbc/execute! ds ["SELECT * FROM address WHERE name = ? AND email = ?"
                   "Stella" "stella@artois.beer"])
```

While the hash map approach -- "query by example" -- is great for equality comparisons, sometimes you need other types of comparisons. For example, you might want to find all the rows where the email address ends in `.beer`:

```
(sql/find-by-keys ds :address ["email LIKE ?" "%.beer"])
;; equivalent to
(jdbc/execute! ds ["SELECT * FROM address WHERE email LIKE ?" "%.beer"])
```

Or you may want to find all the rows where the name is one of a specific set of values:

The default behavior is to return all the columns in each row. You can specify a subset of columns to return using the `:columns` option. It takes a vector and each element of the vector can be:

- a simple keyword representing the column name (`:column-fn` will be applied, if provided),
- a pair of keywords representing the column name and an alias (`:column-fn` will be applied to both, if provided),
- a pair consisting of a string and a keyword, representing a SQL expression and an alias (`:column-fn` will be applied to the alias, if provided).

```
(sql/find-by-keys ds :address {:name "Stella"} {:columns [[:email :address]]})
;; equivalent to
(jdbc/execute! ds ["SELECT email AS address FROM address WHERE name = ?"
                   "Stella"])

(sql/find-by-keys ds :address {:name "Stella"} {:columns [["count(*)" :n]]})
;; equivalent to
(jdbc/execute! ds ["SELECT count(*) AS n FROM address WHERE name = ?"
                   "Stella"])
```

> Note: the SQL string provided for a column is copied exactly as-is into the generated SQL -- you are responsible for ensuring it is legal SQL!

`find-by-keys` supports an `:order-by` option which can specify a vector of column names to sort the results by. Elements may be column names or pairs of a column name and the direction to sort: `:asc` or `:desc`:

```
;; equivalent to
(jdbc/execute! ds ["SELECT * FROM address WHERE name = ? AND email = ? ORDER BY id DESC"
                   "Stella" "stella@artois.beer"])
```

`find-by-keys` also supports basic pagination with `:offset` and `:fetch` options which both accept numeric values and adds `OFFSET ? ROWS FETCH NEXT ? ROWS ONLY` to the generated query. To support MySQL and SQLite, you can specify `:limit` instead `:fetch` which adds `LIMIT ? OFFSET ?` to the generated query instead.

If you want to match all rows in a table -- perhaps with the pagination options in effect -- you can pass the keyword `:all` instead of either a hash map of column names and values or a vector containing a partial `WHERE` clause and parameters.

```
(sql/find-by-keys ds :address :all {:order-by [:id] :offset 5 :fetch 10})
;; equivalent to
(jdbc/execute! ds ["SELECT * FROM address ORDER BY id OFFSET ? ROWS FETCH NEXT ? ROWS ON
```

If no rows match, `find-by-keys` returns `[]`, just like `execute!`.

`aggregate-by-keys`

Added in 1.3.925, this is a wrapper around `find-by-keys` that makes it easier to perform aggregate queries::

```
(sql/aggregate-by-keys ds :address "count(*)" {:name "Stella"
                                               :email "stella@artois.beer"})
```

```
    (get :next_jdbc_aggregate_123))
```

(where `:next_jdbc_aggregate_123` is a unique alias generated by `next.jdbc`, derived from the aggregate expression string).

> Note: the SQL string provided for the aggregate is copied exactly as-is into the generated SQL -- you are responsible for ensuring it is legal SQL!

`get-by-id`

Given a table name (as a keyword) and a primary key value, with an optional primary key column name, execute a query on the database:

```
(sql/get-by-id ds :address 2)
;; equivalent to
(sql/get-by-id ds :address 2 {}) ; empty options map
;; equivalent to
(sql/get-by-id ds :address 2 :id {}) ; empty options map
;; equivalent to
(jdbc/execute-one! ds ["SELECT * FROM address WHERE id = ?" 2])
```

Note that in order to override the default primary key column name (of `:id`), you need to specify both the column name and an options hash map.

If no rows match, `get-by-id` returns `nil`, just like `execute-one!`.

(unqualified) keywords provided. If you are trying to use a table name or column name that is a reserved name in SQL for your database, you will need to tell those functions to quote those names.

The namespace `next.jdbc.quoted` provides five functions that cover the most common types of entity quoting, and a modifier function for quoting dot-separated names (e.g., that include schemas):

- `ansi` -- wraps entity names in double quotes,
- `mysql` -- wraps entity names in back ticks,
- `sql-server` -- wraps entity names in square brackets,
- `oracle` -- an alias for `ansi`,
- `postgres` -- an alias for `ansi`.
- `schema` -- wraps a quoting function to support `dbo.table` style entity names.

These quoting functions can be provided to any of the friendly SQL functions above using the `:table-fn` and `:column-fn` options, in a hash map provided as the (optional) last argument in any call. If you want to provide your own entity naming function, you can do that:

```
(defn snake-case [s] (str/replace s #"-" "_"))

(sql/insert! ds :my-table {:some "data"} {:table-fn snake-case})
```

`next.jdbc` provides `snake-kebab-opts` and `unqualified-snake-kebab-opts` which are hash maps containing `:column-fn` and `:table-fn` that use the `->snake_case` function from the camel-snake-kebab

```
;; column names, in addition, it will perform the reverse transformation
;; on any results, e.g., turning MySQL's :GENERATED_KEY into :generated-key
(sql/insert! ds :my-table {:some "data"} jdbc/snake-kebab-opts)
```

> Note: The entity naming function is passed a string, the result of calling `name` on the keyword passed in. Also note that the default quoting functions do not handle schema-qualified names, such as `dbo.table_name` -- `sql-server` would produce `[dbo.table_name]` from that. Use the `schema` function to wrap the quoting function if you need that behavior, e.g,. `{:table-fn (schema sql-server)}` which would produce `[dbo].[table_name]`.

HugSQL Quick Start

Here's how to get up and running quickly with `next.jdbc` and HugSQL. For more detail, consult the HugSQL documentation. Add the following dependencies to your project (in addition to `com.github.seancorfield/next.jdbc` and whichever JDBC drivers you need):

```
com.layerware/hugsql-core {:mvn/version "0.5.3"}
com.layerware/hugsql-adapter-next-jdbc {:mvn/version "0.5.3"}
```

Check the HugSQL documentation for the latest versions to use!

Write your SQL in `.sql` files that are on the classpath (somewhere under `src` or `resources`). For our purposes, assume a SQL file `db/example.sql` containing your first set of definitions. In your namespace, add these `require`s:

```
[hugsql.core :as hugsql]
```

At program startup you'll need to call these functions (either at the top-level of your namespace or inside your initialization function):

```
;; regular SQL functions
(hugsql/def-db-fns "db/example.sql"
                   {:adapter (adapter/hugsql-adapter-next-jdbc)})

;; development/advanced usage functions that produce a vector containing
;; SQL and parameters that could be passed to jdbc/execute! etc
(hugsql/def-sqlvec-fns "db/example.sql"
                       {:adapter (adapter/hugsql-adapter-next-jdbc)})
```

Those calls will add function definitions to that namespace based on what is in the `.sql` files. Now set up your db-spec and datasource as usual with `next.jdbc`:

```
(def db-spec {:dbtype "h2:mem" :dbname "example"}) ; assumes H2 driver in deps.edn

(def ds (jdbc/get-datasource db-spec))
```

Borrowing from Princess Bride examples from the HugSQL documentation, you can now do things like this:

```
(create-characters-table ds)
;;=> [#:next.jdbc{:update-count 0}]
(insert-character ds {:name "Westley", :specialty "love"})
;;=> 1
```

```
;; add require next.jdbc.result-set :as rs to your ns

(hugsql/def-db-fns "db/example.sql"
                   {:adapter (adapter/hugsql-adapter-next-jdbc
                              {:builder-fn rs/as-maps})})

;; now you'll get qualified as-is hash maps back:
(character-by-id ds {:id 1})
;;=> #:CHARACTERS{:ID 1, :NAME "Westley", :SPECIALTY "love", :CREATED_AT #inst "2019-09-
```

❮ Getting Started

Tips & Tricks ❯

Can you improve this documentation? These fine people already did:
Sean Corfield, alexandrkozyrev & Valtteri Harmainen

Edit on GitHub