`next.jdbc` Options

This section documents all of the options that are supported by all of the functions in `next.jdbc`. Nearly every function accepts an optional hash map as the last argument, that can control many of the behaviors of the library.

The most general options are described first, followed by more specific options that apply only to certain functions.

Datasources and Connections

Although `get-datasource` does not accept options, the "db spec" hash map passed in may contain the following options:

- `:dbtype` -- a string that identifies the type of JDBC database being used,
- `:dbname` -- a string that identifies the name of the actual database being used,
- `:dbname-separator` -- an optional string that can be used to override the `/` or `:` that is normally placed in front of the database name in the JDBC URL,
- `:host` -- an optional string that identifies the IP address or hostname of the server on which the database is running; the default is `"127.0.0.1"`; if `:none` is specified, `next.jdbc` will assume this is for a local database and will omit the host/port segment of the JDBC URL,
- `:host-prefix` -- an optional string that can be used to override the `//` that is normally placed in

or "exotic" database types; if `:none` is specified, `next.jdbc` will omit the port segment of the JDBC URL,

- `:property-separator` -- an optional string that can be used to override the separators used in `next.jdbc.connection/jdbc-url` for the properties (after the initial JDBC URL portion); by default `?` and `&` are used to build JDBC URLs with properties; for SQL Server drivers (both MS and jTDS) `:property-separator ";"` is used, so this option should only be necessary when you are specifying "unusual" databases that `next.jdbc` does not already know about,
- `:classname` -- an optional string that identifies the name of the JDBC driver class to be used for the connection; for common database types, `next.jdbc` knows the default so this should only be needed for "exotic" database types,
- `:user` -- an optional string that identifies the database username to be used when authenticating (NOTE: HikariCP needs `:username` instead – see below),
- `:password` -- an optional string that identifies the database password to be used when authenticating.

If you already have a JDBC URL, you can either specify that string *instead* of a "db spec" hash map or, if you need additional properties passed to the JDBC driver, you can use a hash map containing `:jdbcUrl`, specifying the JDBC URL, and any properties you need as additional keys in the hash map.

Any additional keys provided in the "db spec" will be passed to the JDBC driver as `Properties` when each connection is made. Alternatively, when used with `next.jdbc.connection/->pool`, additional keys correspond to setters called on the pooled connection object.

If you are using HikariCP and `next.jdbc.connection/->pool` to create a connection pooled datasource, you need to provide `:username` for the database username (instead of, or as well as, `:user`).

Any path that calls `get-connection` will accept the following options:

see below),

- `:read-only` -- a `Boolean` that determines whether the operations on this connection should be read-only or not (the default, `false`).
- `:connection` -- a hash map of camelCase properties to set on the `Connection` object after it is created; these correspond to `.set*` methods on the `Connection` class and are set via the Java reflection API (using `org.clojure/java.data`). If `:autoCommit` or `:readOnly` are provided, they will take precedence over the fast, specific options above.

If you need additional options set on a connection, you can either use Java interop to set them directly, or provide them as part of the "db spec" hash map passed to `get-datasource` (although then they will apply to *all* connections obtained from that datasource).

> Note: If `plan`, `execute!`, or `execute-one!` are passed a `DataSource`, a "db spec" hash map, or a JDBC URL string, they will call `get-connection`, so they will accept the above options in those cases.

Generating SQL

Except for `query` (which is simply an alias for `execute!`), all the "friendly" SQL functions accept the following options (in addition to all the options that `plan`, `execute!`, and `execute-one!` can accept):

- `:table-fn` -- the quoting function to be used on the string that identifies the table name, if provided; this also applies to assumed table names when `nav`igating schemas,
- `:column-fn` -- the quoting function to be used on any string that identifies a column name, if provided; this also applies to the reducing function context over `plan` and to assumed foreign key column names when `nav`igating schemas.

which do not support calling `.getGeneratedKeys()` on `PreparedStatement` objects, so you cannot use `:return-generated-keys` to get back the keys -- you must use `RETURNING *`.

In addition, `find-by-keys` accepts the following options (see its docstring for more details):

- `:columns` -- specify one or more columns to `SELECT` to override selecting all columns,
- `:order-by` -- specify one or more columns, on which to sort the results,
- `:top` / `:limit` / `:offset` / `:fetch` to support pagination of results.

In the simple case, the `:columns` option expects a vector of keywords and each will be processed according to `:column-fn`, if provided. A column alias can be specified using a vector pair of keywords and both will be processed according to `:column-fn`, e.g., `[:foo [:bar :quux]]` would expand to `foo, bar AS quux`. You can also specify the first element of the pair as a string which will be used as-is in the generated SQL, e.g., `[:foo ["COUNT(*)" :total]]` would expand to `foo, COUNT(*) AS total`. In the latter case, the alias keyword will still be processed according to `:column-fn` but the string will be untouched -- you are responsible for any quoting and/or other formatting that might be required to produce a valid SQL expression.

> Note: `get-by-id` accepts the same options as `find-by-keys` but it will only ever produce one row, as a hash map, so sort order and pagination are less applicable, although `:columns` may be useful.

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` except that `:columns` may not be specified (since it is used to add the aggregate to the query).

Generating Rows and Result Sets

only needs that if it actually has to realize a row) but most generation functions will implement both for ease of use.

- `:label-fn` -- if `:builder-fn` is specified as one of `next.jdbc.result-set`'s `as-modified-*` builders, this option must be present and should specify a string-to-string transformation that will be applied to the column label for each returned column name.
- `:qualifier-fn` -- if `:builder-fn` is specified as one of `next.jdbc.result-set`'s `as-modified-*` builders, this option should specify a string-to-string transformation that will be applied to the table name for each returned column name. It will be called with an empty string if the table name is not available. It can be omitted for the `as-unqualified-modified-*` variants.
- `:column-fn` -- if present, applied to each column name before looking up the column in the `ResultSet` to get that column's value.

In addition, `execute!` accepts the `:multi-rs true` option to return multiple result sets -- as a vector of result sets.

> Note: Subject to the caveats above about `:builder-fn`, that means that `plan`, `execute!`, `execute-one!`, and the "friendly" SQL functions will all accept these options for generating rows and result sets.

Datafying & Navigating Rows and Result Sets

Any function that produces a result set will accept the following options that modify the behavior of `datafy` and `nav` applied to the rows in that result set:

- `:schema` -- override the conventions for identifying foreign keys and the related (primary) keys in the tables to which they refer, on a per table/column basis; can also be used to indicate a fk relationship is

See `datafy`, `nav`, and `:schema` for more details.

Statements & Prepared Statements

Any function that creates a `Statement` or a `PreparedStatement` will accept the following options (see below for additional options for `PreparedStatement`):

- `:concurrency` -- a keyword that specifies the concurrency level: `:read-only`, `:updatable`,
- `:cursors` -- a keyword that specifies whether cursors should be closed or held over a commit: `:close`, `:hold`,
- `:fetch-size` -- an integer that guides the JDBC driver in terms of how many rows to fetch at once; the actual behavior of specifying `:fetch-size` is database-specific: some JDBC drivers use a zero or negative value here to trigger streaming of result sets -- other JDBC drivers require this to be positive for streaming and may require additional options to be set on the connection *as well*,
- `:max-rows` -- an integer that tells the JDBC driver to limit result sets to this many rows,
- `:result-type` -- a keyword that affects how the `ResultSet` can be traversed: `:forward-only`, `:scroll-insensitive`, `:scroll-sensitive`,
- `:timeout` -- an integer that specifies the (query) timeout allowed for SQL operations, in seconds. See **Handling Timeouts** in **Tips & Tricks** for more details on this and other possible timeout settings.
- `:statement` -- a hash map of camelCase properties to set on the `Statement` or `PreparedStatement` object after it is created; these correspond to `.set*` methods on the `Statement` class (which `PreparedStatement` inherits) and are set via the Java reflection API (using `org.clojure/java.data`). If `:fetchSize`, `:maxRows`, or `:queryTimeout` are provided, they will take precedence over the fast, specific options above.

> `:result-type` with one of the scroll values (and so you must also specify `:concurrency`).

Any function that creates a `PreparedStatement` will additionally accept the following options:

- `:return-keys` -- a truthy value asks that the JDBC driver to return any generated keys created by the operation; it can be `true` or it can be a vector of keywords identifying column names that should be returned.

Not all databases or drivers support all of these options, or all values for any given option. If `:return-keys` is a vector of column names and that is not supported, `next.jdbc` will attempt a generic "return generated keys" option instead. If that is not supported, `next.jdbc` will fall back to a regular SQL operation. If other options are not supported, you may get a `SQLException`. You may need to use `RETURNING *` on `INSERT` statements instead of using `:return-keys` with some database drivers.

> Note: If `plan`, `execute!`, or `execute-one!` are passed a `DataSource`, a "db spec" hash map, or a JDBC URL string, they will call `prepare` to create a `PreparedStatement`, so they will accept the above options in those cases.

In addition to the above, `next.jdbc/execute-batch!` (which may create a `PreparedStatement` if you pass in a SQL string and either a `Connection` or `DataSource`) accepts an options hash map that can also contain the following:

- `:batch-size` -- an integer that determines how to partition the parameter groups for submitting to the database in batches,
- `:large` -- a Boolean flag that indicates whether the batch will produce large update counts (`long` rather than `int` values),
- `:return-generated-keys` -- a Boolean flag that indicates whether `.getGeneratedKeys` should be

Transactions

The `transact` function and `with-transaction` (`+options`) macro accept the following options:

- `:isolation` -- a keyword that identifies the isolation to be used for this transaction: `:none`, `:read-committed`, `:read-uncommitted`, `:repeatable-read`, or `:serializable`; these represent increasingly strict levels of transaction isolation and may not all be available depending on the database and/or JDBC driver being used,
- `:read-only` -- a `Boolean` that indicates whether the transaction should be read-only or not (the default),
- `:rollback-only` -- a `Boolean` that indicates whether the transaction should commit on success (the default) or rollback.

Plan Selection

The `next.jdbc.plan/select!` function accepts the following specific option:

- `:into` -- a data structure into which the selected result from a `plan` operation are poured; by default this is `[]`; could be any value that is acceptable as the first argument to `into`, subject to `into` accepting the sequence of values produced by the `plan` reduction.

❮ Transactions datafy, nav, and :schema ❯