Tips & Tricks

This page contains various tips and tricks that make it easier to use `next.jdbc` with a variety of databases. It is mostly organized by database, but there are a few that are cross-database and those are listed first.

CLOB & BLOB SQL Types

Columns declared with the `CLOB` or `BLOB` SQL types are typically rendered into Clojure result sets as database-specific custom types but they should implement `java.sql.Clob` or `java.sql.Blob` (as appropriate). In general, you can only read the data out of those Java objects during the current transaction, which effectively means that you need to do it either inside the reduction (for `plan`) or inside the result set builder (for `execute!` or `execute-one!`). If you always treat these types the same way for all columns across the whole of your application, you could simply extend `next.jdbc.result-set/ReadableColumn` to `java.sql.Clob` (and/or `java.sql.Blob`). Here's an example for reading `CLOB` into a `String`:

```
(extend-protocol rs/ReadableColumn
  java.sql.Clob
  (read-column-by-label [^java.sql.Clob v _]
    (with-open [rdr (.getCharacterStream v)] (slurp rdr)))
  (read-column-by-index [^java.sql.Clob v _2 _3]
    (with-open [rdr (.getCharacterStream v)] (slurp rdr))))
```

There is a helper in `next.jdbc.result-set` to make this easier -- `clob->string`:

```
    (rs/clob->string v))
  (read-column-by-index [^java.sql.Clob v _2 _3]
    (rs/clob->string v)))
```

As noted in Result Set Builders, there is also `clob-column-reader` that can be used with the `as-*-adapter` result set builder functions.

No helper or column reader is provided for `BLOB` data since it is expected that the semantics of any given binary data will be application specific. For a raw `byte[]` you could probably use:

```
(.getBytes v 1 (.length v)) ; BLOB has 1-based byte index!
```

Consult the java.sql.Blob documentation for more ways to process it.

> Note: the standard MySQL JDBC driver seems to return `BLOB` data as `byte[]` instead of `java.sql.Blob`.

Exceptions

A lot of JDBC operations can fail with an exception. JDBC 4.0 has a well-defined hierarchy of exception types and you can often catch a specific type of exception to do useful handling of various error conditions that you might "expect" when working with a database.

A good example is SQLIntegrityConstraintViolationException which typically represents an index/key constraint violation such as a duplicate primary key insertion attempt.

However, like some other areas when dealing with JDBC, the reality can be very database-specific. Some

The `java.sql.SQLException` class provides `.getErrorCode()` and `.getSQLState()` methods but the values returned by those are explicitly vendor-specific (error code) or only partly standardized (state). In theory, the SQL state should follow either the X/Open (Open Group) or ANSI SQL 2003 conventions, both of which were behind paywalls(!). The most complete public listing is probably the IBM DB2 SQL State document. See also this Stack Overflow post about SQL State for more references and links. Not all database drivers follow either of these conventions for SQL State so you may still have to consult your vendor's specific documentation.

All of this makes writing *generic* error handling, that works across multiple databases, very hard indeed. You can't rely on the JDBC `SQLException` hierarchy; you can sometimes rely on a subset of SQL State values.

Handling Timeouts

JDBC provides a number of ways in which you can decide how long an operation should run before it times out. Some of these timeouts are specified in seconds and some are in milliseconds. Some are handled via connection properties (or JDBC URL parameters), some are handled via methods on various JDBC objects.

Here's how to specify various timeouts using `next.jdbc`:

- `connectTimeout` -- can be specified via the "db-spec" hash map or in a JDBC URL, it is the number of **milliseconds** that JDBC should wait for the initial (socket) connection to complete. Database-specific (may be MySQL only?).
- `loginTimeout` -- can be set via `.setLoginTimeout()` on a `DriverManager` or `DataSource`, it is the number of **seconds** that JDBC should wait for a connection to the database to be made. `next.jdbc` exposes this on the `javax.sql.DataSource` object it reifies from calling `get-datasource` on a "db-spec" hash map or JDBC URL string.

passed to any function that may construct a `Statement` or `PreparedStatement`.

- `socketTimeout` -- can be specified via the "db-spec" hash map or in a JDBC URL, it is the number of milliseconds that JDBC should wait for socket operations to complete. Database-specific (MS SQL Server and MySQL support this, other databases may too).

Examples:

```
;; connectTimeout / socketTimeout via db-spec:
(def db-spec {:dbtype "mysql" :dbname "example" :user "root" :password "secret"
              ;; milliseconds:
              :connectTimeout 60000 :socketTimeout 30000}))

;; socketTimeout via JDBC URL:
(def db-url (str "jdbc:sqlserver://localhost;user=sa;password=secret"
                 ;; milliseconds:
                 ";database=model;socketTimeout=10000"))

;; loginTimeout via DataSource:
(def ds (jdbc/get-datasource db-spec))
(.setLoginTimeout ds 20) ; seconds

;; queryTimeout via options:
(jdbc/execute! ds ["select * from some_table"] {:timeout 5}) ; seconds

;; queryTimeout via method call:
(let [ps (jdbc/prepare ds ["select * from some_table"])]
  (.setQueryTimeout ps 10) ; seconds
  (jdbc/execute! ps))
```

overhead of realizing rows from the `ResultSet` into Clojure data structures if your reducing function uses only functions that get column values by name. If you perform any function on the row that would require an actual hash map or a sequence, the row will be realized into a full Clojure hash map via the builder function passed in the options (or via `next.jdbc.result-set/as-maps` by default).

One of the benefits of reducing over `plan` is that you can stream very large result sets, very efficiently, without having the entire result set in memory (assuming your reducing function doesn't build a data structure that is too large!). See the tips below on **Streaming Result Sets**. If you want to process a `plan` result purely for side-effects, without producing a result, you can use `run!` instead of `reduce`:

```
(run! process-row (jdbc/plan ds ...))
```

`run!` is based on `reduce` and `process-row` here takes just one argument -- the row -- rather than the usual reducing function that takes two.

The result of `plan` is also foldable in the clojure.core.reducers sense. While you could use `execute!` to produce a vector of fully-realized rows as hash maps and then fold that vector (Clojure's vectors support fork-join parallel reduce-combine), that wouldn't be possible for very large result sets. If you fold the result of `plan`, the result set will be partitioned and processed using fork-join parallel reduce-combine. Unlike reducing over `plan`, each row **is** realized into a Clojure data structure and each batch is forked for reduction as soon as that many rows have been realized. By default, `fold`'s batch size is 512 but you can specify a different value in the 4-arity call. Once the entire result set has been read, the last (partial) batch is forked for reduction. The combining operations are forked and interleaved with the reducing operations, so the order (of forked tasks) is batch-1, batch-2, combine-1-2, batch-3, combine-1&2-3, batch-4, combine-1&2&3-4, etc. The amount of parallelization you get will depend on many factors including the number of processors, the speed of your reducing function, the speed of your combining function, and the

Times, Dates, and Timezones

Working with dates and timezones in databases can be confusing, as you are working at the intersection between the database, the JDBC library and the date library that you happen to be using. A good rule of thumb is to keep timezone-related logic as simple as possible. For example, with Postgres we recommend always storing dates in a Postgres `TIMESTAMP` (without time zone) column, storing all such timestamps in UTC, and applying your time zone logic separately using application logic. The `TIMESTAMP WITH TIME ZONE` column type in Postgres stores its date in UTC anyhow, and applications that need to deal with time zones typically require richer functionality than simply adjusting the time zone to wherever the database happens to be hosted. Treat time zone related logic as an application concern, and keep stored dates in UTC.

For example, for a developer using `clojure.java-time`, saving `(java-time/instant)` in a timestamp column (and doing any timezone adjustment elsewhere) is a good way to minimize long term confusion.

> Original text contributed by Denis McCarthy; in addition: I generally recommend not only using UTC everywhere but also setting your database *and your servers* to all be in the UTC timezones, to avoid the possibly of incorrect date/time translations -- Sean Corfield.

MS SQL Server

In MS SQL Server, the generated key from an insert comes back as `:GENERATED_KEYS`.

By default, you won't get table names as qualifiers with Microsoft's JDBC driver (you might with the jTDS drive -- I haven't tried that recently). See this MSDN forum post about `.getTableName()` for details. According to one of the answers posted there, if you specify `:result-type` and `:concurrency` in the

MS SQL Server supports execution of multiple statements when surrounded by `begin` / `end` and can return multiple result sets, when requested via `:multi-rs true` on `execute!`.

```
(jdbc/execute! db-spec ["begin select * from table1; select * from table2; end"] {:multi
;; vector of result sets:
=> [[{.. table1 row ..} {.. table1 row ..}]
    [{.. table2 row ..} {.. table2 row ..} {..}]]
```

Batch Statements

Even when using `next.jdbc/execute-batch!`, Microsoft's JDBC driver will still send multiple insert statements to the database unless you specify `:useBulkCopyForBatchInsert true` as part of the db-spec hash map or JDBC URL when the datasource is created.

To use this feature your Microsoft's JDBC driver should be at least version 9.2 and you can use only limited set of data types. For example if you use `inst` to bulk insert smalldatetime value driver will revert to old (slow) behavior. For more details see Using bulk copy API for batch insert operation and Release notes for JDBC drivers.

MySQL & MariaDB

In MySQL, the generated key from an insert comes back as `:GENERATED_KEY`. In MariaDB, the generated key from an insert comes back as `:insert_id`.

MySQL generally stores tables as files so they are case-sensitive if your O/S is (Linux) or case-insensitive if your O/S is not (Mac, Windows) but the column names are generally case-insensitive. This can matter when

because of this.

It's also worth noting that column comparisons are case-insensitive so `WHERE foo = 'BAR'` will match `"bar"` or `"BAR"` etc.

MySQL has a connection option, `:allowMultiQueries true`, that allows you to pass multiple SQL statements in a single operation and can return multiple result sets, when requested via `:multi-rs true`.

```
(def db-spec {:dbtype "mysql" .. :allowMultiQueries true})
;; equivalent to allowMultiQueries=true in the JDBC URL
(jdbc/execute! db-spec ["select * from table1; select * from table2"] {:multi-rs true})
;; vector of result sets:
=> [[{.. table1 row ..} {.. table1 row ..}]
    [{.. table2 row ..} {.. table2 row ..} {..}]]
```

Compare this with MS SQL Server above: MySQL does not support `begin` / `end` here. This is not the default behavior because allowing multiple statements in a single operation is generally considered a bit of a risk as it can make it easier for SQL injection attacks to be performed.

Batch Statements

Even when using `next.jdbc/execute-batch!`, MySQL will still send multiple statements to the database unless you specify `:rewriteBatchedStatements true` as part of the db-spec hash map or JDBC URL when the datasource is created.

Streaming Result Sets

> Note: it's possible that other options may be required as well -- I have not verified this yet -- see, for example, the additional options PostgreSQL requires, below.

Oracle

Ah, dear old Oracle! Over the years of maintaining `clojure.java.jdbc` and now `next.jdbc`, I've had all sorts of bizarre and non-standard behavior reported from Oracle users. The main issue I'm aware of with `next.jdbc` is that Oracle's JDBC drivers all return an empty string from `ResultSetMetaData.getTableName()` so you won't get qualified keywords in the result set hash maps. Sorry!

An important performance issue to be aware of with Oracle's JDBC driver is that the default fetch size is just 10 records. If you are working with large datasets, you will either need to either specify `:prefetch` in your db-spec hash map with a suitable value (say 1,000 or larger), or specify `&prefetch=` in your JDBC URL string. If you want to keep the default, you can change it on a per-statement basis by specifying `:fetch-size` as an option to `execute!` etc.

If you are using the 10g or later JDBC driver and you try to execute DDL statements that include SQL entities that start with a `:` (such as `:new` or `:old`), they will be treated as bindable parameter references if you use a `PreparedStatement` to execute them. Since that's the default for `execute!` etc, it means that you will likely get an error like the following:

```
Missing IN or OUT parameter at index:: 1
```

You will need to use `next.jdbc.prepare/statement` to create a `Statement` object and then call

PostgreSQL

As you can see in this section (and elsewhere in this documentation), the PostgreSQL JDBC driver has a number of interesting quirks and behaviors that you need to be aware of. Although accessing PostgreSQL via JDBC is the most common approach, there is also a non-JDBC Clojure/Java driver for PostgreSQL called PG2 which supports JSON operations natively (see below for what's required for JDBC), as well as supporting Java Time natively (see the section above about **Times, Dates, and Timezones**), and it also quite a bit faster than using JDBC.

When you use `:return-keys true` with `execute!` or `execute-one!` (or you use `insert!`), PostgreSQL returns the entire inserted row (unlike nearly every other database that just returns any generated keys!).

The default result set builder for `next.jdbc` is `as-qualified-maps` which uses the `.getTableName()` method on `ResultSetMetaData` to qualify the columns in the result set. While some database drivers have this information on hand from the original SQL operation, PostgreSQL's JDBC driver does not and it performs an extra SQL query to fetch table names the first time this method is called for each query. If you want to avoid those extra queries, and you can live with unqualified column names, you can use `as-unqualified-maps` as the result set builder instead.

If you have a query where you want to select where a column is `IN` a sequence of values, you can use `col = ANY(?)` with a native array of the values instead of `IN (?,?,?,,,?)` and a sequence of values. **Be aware of PostgreSQL bug 17822 which can cause pathological performance when the array has a single element!** If you think you might have a single-element array, consider using `UNNEST` and `IN` instead.

What does this mean for your use of `next.jdbc`? In `plan`, `execute!`, and `execute-one!`, you can use `col = ANY(?)` in the SQL string and a single primitive array parameter, such as `(int-array [1 2 3 4])`.

Batch Statements

Even when using `next.jdbc/execute-batch!`, PostgreSQL will still send multiple statements to the database unless you specify `:reWriteBatchedInserts true` as part of the db-spec hash map or JDBC URL when the datasource is created.

Streaming Result Sets

You can get PostgreSQL to stream very large result sets (when you are reducing over `plan`) by setting the following options:

- `:auto-commit false` -- when opening the connection
- `:fetch-size 4000, :concurrency :read-only, :cursors :close, :result-type :forward-only` -- when running `plan` (or when creating a `PreparedStatement`).

Working with Arrays

ResultSet protocol extension to read SQL arrays as Clojure vectors.

```
(extend-protocol rs/ReadableColumn
  Array
  (read-column-by-label [^Array v _]    (vec (.getArray v)))
  (read-column-by-index [^Array v _ _]  (vec (.getArray v))))
```

Insert and read vector example:

```
create table example(
  tags varchar[]
);
```

```
(execute-one! db-spec
  ["insert into example(tags) values (?)"
    (into-array String ["tag1" "tag2"])])

(execute-one! db-spec
  ["select * from example limit 1"])

;; => #:example{:tags ["tag1" "tag2"]}
```

> Note: PostgreSQL JDBC driver supports only 7 primitive array types, but not array types like `UUID[]` - PostgreSQL™ Extensions to the JDBC API.

Working with Date and Time

In addition, if you want `java.time.Instant`, `java.time.LocalDate`, and `java.time.LocalDateTime` to be automatically converted to SQL data types, requiring `next.jdbc.date-time` will enable those as well (by extending `SettableParameter` for you).

`next.jdbc.date-time` also includes functions that you can call at application startup to extend `ReadableColumn` to either return `java.time.Instant` or `java.time.LocalDate` / `java.time.LocalDateTime` (as well as a function to restore the default behavior of returning `java.sql.Date` and `java.sql.Timestamp`).

Working with Interval

Postgres has a nonstandard SQL type Interval that is implemented in the Postgres driver as the `org.postgresql.util.PGInterval` type. In many cases you would want to work with intervals as `java.time.Duration` type by default.

You can support `Duration` instances by extending `SettableParameter` to the `java.time.Duration` type. Conversely you can support converting PGIntervals back to Durations by extending `ReadableColumn` to the `org.postgresql.util.PGInterval` type.

```
(import '[org.postgresql.util PGInterval])
(import '[java.sql PreparedStatement])
(import '[java.time Duration])
(require '[next.jdbc.result-set :as rs])
(require '[next.jdbc.prepare :as p])

(defn ->pg-interval
  "Takes a Dudration instance and converts it into a PGInterval
```

```
(extend-protocol p/SettableParameter
  ;; Convert durations to PGIntervals before inserting into db
  java.time.Duration
  (set-parameter [^java.time.Duration v ^PreparedStatement s ^long i]
    (.setObject s i (->pg-interval v))))


(defn <-pg-interval
  "Takes a PGInterval instance and converts it into a Duration
   instance. Ignore sub-second units."
  [^org.postgresql.util.PGInterval interval]
  (-> Duration/ZERO
      (.plusSeconds (.getSeconds interval))
      (.plusMinutes (.getMinutes interval))
      (.plusHours (.getHours interval))
      (.plusDays (.getDays interval))))

(extend-protocol rs/ReadableColumn
  ;; Convert PGIntervals back to durations
  org.postgresql.util.PGInterval
  (read-column-by-label [^org.postgresql.util.PGInterval v _]
    (<-pg-interval v))
  (read-column-by-index [^org.postgresql.util.PGInterval v _2 _3]
    (<-pg-interval v)))
```

Working with Enumerated Types

PostgreSQL has a SQL extension for defining enumerated types and the default `set-parameter`

```
CREATE TYPE language AS ENUM('en','fr','de');

CREATE TABLE person (
  ...
  speaks language NOT NULL,
  ...
);
```

```
(require '[next.jdbc.sql :as sql]
         '[next.jdbc.types :refer [as-other]])

(sql/insert! ds :person {:speaks (as-other "fr")})
```

That call produces a vector `["fr"]` with metadata that implements `set-parameter` such that `.setObject()` is called with `java.sql.Types/OTHER` which allows PostgreSQL to "convert" the string `"fr"` to the corresponding `language` enumerated type value.

Working with JSON and JSONB

PostgreSQL has good support for storing, querying and manipulating JSON data. Basic Clojure data structures (lists, vectors, and maps) transform pretty well to JSON data. With a little help `next.jdbc` can automatically convert Clojure data to JSON and back for us.

> Note: some PostgreSQL JSONB operators have a `?` in them which conflicts with the standard parameter placeholder in SQL. You can write the JSONB operators by doubling up the `?`, e.g., `??|` instead of just `?|`. See PostgreSQL JSONB operators for more detail.

```
(require '[jsonista.core :as json])

;; :decode-key-fn here specifies that JSON-keys will become keywords:
(def mapper (json/object-mapper {:decode-key-fn keyword}))
(def ->json json/write-value-as-string)
(def <-json #(json/read-value % mapper))
```

Next we create helper functions to transform Clojure data to and from PostgreSQL Objects containing JSON:

```
(import '(org.postgresql.util PGobject))

(defn ->pgobject
  "Transforms Clojure data to a PGobject that contains the data as
  JSON. PGObject type defaults to `jsonb` but can be changed via
  metadata key `:pgtype`"
  [x]
  (let [pgtype (or (:pgtype (meta x)) "jsonb")]
    (doto (PGobject.)
      (.setType pgtype)
      (.setValue (->json x)))))

(defn <-pgobject
  "Transform PGobject containing `json` or `jsonb` value to Clojure data."
  [^PGobject v]
  (let [type  (.getType v)
        value (.getValue v)]
    (if (#{"jsonb" "json"} type)
      (some-> value <-json (with-meta {:pgtype type}))
```

protocols to make the conversion between clojure data and PGobject JSON automatic:

```
(require '[next.jdbc.prepare :as prepare])
(require '[next.jdbc.result-set :as rs])

(import  '[java.sql PreparedStatement])

(set! *warn-on-reflection* true)

;; if a SQL parameter is a Clojure hash map or vector, it'll be transformed
;; to a PGobject for JSON/JSONB:
(extend-protocol prepare/SettableParameter
  clojure.lang.IPersistentMap
  (set-parameter [m ^PreparedStatement s i]
    (.setObject s i (->pgobject m)))

  clojure.lang.IPersistentVector
  (set-parameter [v ^PreparedStatement s i]
    (.setObject s i (->pgobject v))))

;; if a row contains a PGobject then we'll convert them to Clojure data
;; while reading (if column is either "json" or "jsonb" type):
(extend-protocol rs/ReadableColumn
  org.postgresql.util.PGobject
  (read-column-by-label [^org.postgresql.util.PGobject v _]
    (<-pgobject v))
  (read-column-by-index [^org.postgresql.util.PGobject v _2 _3]
    (<-pgobject v)))
```

```
create table demo (
  id          serial primary key,
  doc_jsonb   jsonb,
  doc_json    json
)
```

We can now insert Clojure data into json and jsonb fields:

```
(require '[next.jdbc :as jdbc])
(require '[next.jdbc.sql :as sql])

(def db { ...db-spec here... })
(def ds (jdbc/get-datasource db))

(def test-map
  {:some-key "some val" :nested {:a 1} :null-val nil :vector [1 2 3]})

(def data1
  {:doc_jsonb test-map
   :doc_json  (with-meta test-map {:pgtype "json"})})

(sql/insert! ds :demo data1)

(def test-vector
    [{:a 1} nil 2 "lalala" []])

(def data2
    {:doc_jsonb test-vector
```

And those columns are nicely transformed into Clojure data when querying:

```
(sql/get-by-id ds :demo 1)
=> #:demo{:id 1,
          :doc_json
          {:some-key "some val",
           :nested {:a 1},
           :vector [1 2 3],
           :null-val nil},
          :doc_jsonb
          {:some-key "some val",
           :nested {:a 1},
           :vector [1 2 3],
           :null-val nil}}

(sql/get-by-id ds :demo 2)
=> #:demo{:id 2,
          :doc_json [{:a 1} nil 2 "lalala" []],
          :doc_jsonb [{:a 1} nil 2 "lalala" []]}

;; Query by value of JSON field 'some-key'
(sql/query ds [(str "select id, doc_jsonb::json->'nested' as foo"
                    "  from demo where doc_jsonb::json->>'some-key' = ?")
               "some val"])
=> [{:demo/id 1, :foo {:a 1}}]
```

Using HoneySQL with JSON and JSONB

in the HoneySQL documentation for more details.

JSON or JSONB?

- A `json` column stores JSON data as strings (reading and writing is fast but manipulation is slow, field order is preserved)
- A `jsonb` column stores JSON data in binary format (manipulation is significantly faster but reading and writing is a little slower)

If you're unsure whether you want to use json or jsonb, use jsonb.

SQLite

SQLite supports both `bool` and `bit` column types but, unlike pretty much every other database out there, it yields `0` or `1` as the column value instead of `false` or `true`. This means that with SQLite alone, you can't just rely on `bool` or `bit` columns being treated as truthy/falsey values in Clojure.

You can work around this using a builder that handles reading the column directly as a `Boolean`:

```
(import java.sql ResultSet ResultSetMetaData)

(jdbc/execute! ds ["select * from some_table"]
               {:builder-fn (rs/builder-adapter
                             rs/as-maps
                             (fn [builder ^ResultSet rs ^Integer i]
                               (let [rsm ^ResultSetMetaData (:rsmeta builder)]
                                 (rs/read-column-by-index
                                   (if (#{"BIT" "BOOL" "BOOLEAN"} (.getColumnTypeName rs
```

If you are using `plan`, you'll most likely be accessing columns by just the label (as a keyword) and avoiding the result set building machinery completely. In such cases, you'll still get `bool` and `bit` columns back as `0` or `1` and you'll need to explicitly convert them on a per-column basis since you should know which columns need converting:

```
(reduce (fn [acc row]
          (conj acc (-> (select-keys row [:name :is_active])
                        (update :is_active pos?))))
        []
        (jdbc/plan ds ["select * from some_table"]))
```

See also `datafy`, `nav`, and `:schema` > **SQLite** for additional caveats on the `next.jdbc.datafy` namespace when using SQLite.

❮ Friendly SQL Functions Result Set Builders ❯