`datafy`, `nav`, and the `:schema` option

Clojure 1.10 introduced a new namespace, `clojure.datafy`, and two new protocols (`Datafiable` and `Navigable`) that allow for generalized, lazy navigation around data structures. Cognitect also released REBL (now Nubank's Morse) -- a graphical, interactive tool for browsing Clojure data structures, based on the new `datafy` and `nav` functions.

Shortly after REBL's release, I added experimental support to `clojure.java.jdbc` for `datafy` and `nav` that supported lazy navigation through result sets into foreign key relationships and connected rows and tables. `next.jdbc` bakes that support into result sets produced by `execute!` and `execute-one!`.

In addition to `datafy` and `nav` support in the result sets, as of version 1.0.462, there is a `next.jdbc.datafy` namespace that can be required to extend these protocols to a number of JDBC object types. See **JDBC Datafication** near the end of this page for more detail of this.

Additional tools that understand `datafy` and `nav` include Portal and Reveal.

The `datafy` / `nav` Lifecycle on Result Sets

Here's how the process works, for result sets produced by `next.jdbc`:

- `execute!` and `execute-one!` produce result sets containing rows that are `Datafiable`,
- Tools like Portal, Reveal, and Morse can call `datafy` on result sets to render them as "pure data"

- If a column in a row represents a foreign key into another table, calling `nav` will fetch the related row(s),
- Those can in turn be `datafy`'d and `nav`'d to continue drilling down through connected data in the database.

In addition to `execute!` and `execute-one!`, you can call `next.jdbc.result-set/datafiable-result-set` on any `ResultSet` object to produce a result set whose rows are `Datafiable`. Inside a reduction over the result of `plan`, you can call `next.jdbc.result-set/datafiable-row` on a row to produce a `Datafiable` row. That will realize the entire row, including generating column names using the row builder specified (or `as-maps` by default).

Identifying Foreign Keys

By default, `next.jdbc` assumes that a column named `<something>id` or `<something>_id` is a foreign key into a table called `<something>` with a primary key called `id`. As an example, if you have a table `address` which has columns `id` (the primary key), `name`, `email`, etc, and a table `contact` which has various columns including `addressid`, then if you retrieve a result set based on `contact`, call `datafy` on it and then "drill down" into the columns, when `(nav row :contact/addressid v)` is called (where `v` is the value of that column in that row) `next.jdbc`'s implementation of `nav` will fetch a single row from the `address` table, identified by `id` matching `v`.

You can override this default behavior for any column in any table by providing a `:schema` option that is a hash map whose keys are column names (usually the table-qualified keywords that `next.jdbc` produces by default) and whose values are table-qualified keywords, optionally wrapped in vectors, that identity the name of the table to which that column is a foreign key and the name of the key column within that table.

As of 1.3.909, you can also override this behavior via the `:schema-opts` option. This is a hash map whose

- `:pk` -- a string used instead of `id` for the primary key column in the target table,
- `:pk-fn` -- a function that takes the table name and the value of `:pk` and returns the name of the primary key column in the target table, instead of just using the value of `:pk` (the default is effectively `(constantly <pk>)`).

For `:fk-suffix`, the `_` is still permitted and optional in the column name, so if you specified `:schema-opts {:fk-suffix "fk"}` then `addressfk` and `address_fk` would both be treated as foreign keys into the `address` table.

Note: as of 1.3.939, `-` is permitted in key names (in addition to `_`) so that kebab result set builders work as expected.

The `:pk-fn` can use the table name to determine the primary key column name for exceptions to the `:pk` value. For example, if you have a table `address` with a primary key column `address_id` instead of `id`, you could use:

```
:pk-fn (fn [table pk]
         (if (= "address" table)
           "address_id"
           pk))
```

The default behavior in the example above is equivalent to this `:schema` value:

```
                {:schema {:contact/addressid :address/id}})
```

or these `:schema-opts` values:

```
(jdbc/execute! ds
               ["select * from contact where city = ?" "San Francisco"]
               ;; a one-to-one or many-to-one relationship
               {:schema-opts {:fk-suffix "id" :pk "id"
                              :pk-fn (constantly "id")}})
```

If you had a table to track the valid/bouncing status of email addresses over time, `:deliverability`, where `email` is the non-unique key, you could provide automatic navigation into that using:

```
(jdbc/execute! ds
               ["select * from contact where city = ?" "San Francisco"]
               ;; one-to-many or many-to-many
               {:schema {:contact/addressid :address/id
                         :address/email [:deliverability/email]}})
```

Since this relies on a foreign key that does not follow a standard suffix pattern, there is no comparable `:schema-opts` version. In addition, the `:schema-opts` approach cannot designate a one-to-many or many-to-many relationship.

When you indicate a `*-to-many` relationship, by wrapping the foreign table/key in a vector, `next.jdbc`'s implementation of `nav` will fetch a multi-row result set from the target table.

structure from that entity map.

Behind The Scenes

Making rows datafiable is implemented by adding metadata to each row with a key of `clojure.core.protocols/datafy` and a function as the value. That function closes over the connectable and options passed in to the `execute!` or `execute-one!` call that produced the result set containing those rows.

When called (`datafy` on a row), it adds metadata to the row with a key of `clojure.core.protocols/nav` and another function as the value. That function also closes over the connectable and options passed in.

When that is called (`nav` on a row, column name, and column value), if a `:schema` entry exists for that column or it matches the convention described above (either by default or via `:schema-opts`), then it will fetch row(s) using `next.jdbc` 's `Executable` functions `-execute-one` or `-execute-all` , passing in the connectable and options closed over.

The protocol `next.jdbc.result-set/DatafiableRow` has a default implementation of `datafiable-row` for `clojure.lang.IObj` that just adds the metadata to support `datafy` . There is also an implementation baked into the result set handling behind `plan` so that you can call `datafiable-row` directly during reduction and get a fully-realized row that can be `datafy` 'd (and then `nav` igated).

In addition, you can call `next.jdbc.result-set/datafiable-result-set` on any `ResultSet` object and get a fully realized, datafiable result set created using any of the result set builders.

JDBC Datafication

`org.clojure/java.data`), with some additions as described below:

- `java.sql.Connection` -- datafies as a bean; The `:metaData` property is a `java.sql.DatabaseMetaData`, which is also datafiable.
- `DatabaseMetaData` -- datafies as a bean, with an additional `:all-tables` property (that is a dummy object); six properties are navigable to produce fully-realized datafiable result sets:
  - `all-tables` -- produced from `(.getTables this nil nil nil nil)`, this is all the tables and views available from the connection that produced the database metadata,
  - `catalogs` -- produced from `(.getCatalogs this)`
  - `clientInfoProperties` -- all the client properties that the database driver supports,
  - `schemas` -- produced from `(.getSchemas this)`,
  - `tableTypes` -- produced from `(.getTableTypes this)`,
  - `typeInfo` -- produced from `(.getTypeInfo this)`.
- `ParameterMetaData` -- datafies as a vector of parameter descriptions; each parameter hash map has: `:class` (the name of the parameter class -- JVM), `:mode` (one of `:in`, `:in-out`, or `:out`), `:nullability` (one of: `:null`, `:not-null`, or `:unknown`), `:precision`, `:scale`, `:type` (the name of the parameter type -- SQL), and `:signed` (Boolean).
- `ResultSet` -- datafies as a bean; if the `ResultSet` has an associated `Statement` and that in turn has an associated `Connection` then an additional key of `:rows` is provided which is a datafied result set, from `next.jdbc.result-set/datafiable-result-set` with default options. This is provided as a convenience, purely for datafication of other JDBC data types -- in normal `next.jdbc` usage, result sets are datafied under full user control.
- `ResultSetMetaData` -- datafies as a vector of column descriptions; each column hash map has: `:auto-increment`, `:case-sensitive`, `:catalog`, `:class` (the name of the column class -- JVM),

- `Statement` -- datafies as a bean.

See the Java documentation for these JDBC types for further details on what all the properties from each of these classes mean and which are `int`, `String`, or some other JDBC object type.

In addition, requiring this namespace will affect how `next.jdbc.result-set/metadata` behaves inside the reducing function applied to the result of `plan`. Without this namespace loaded, that function will return a raw `ResultSetMetaData` object (which must not leak outside the reducing function). With this namespace loaded, that function will, instead, return a Clojure data structure describing the columns in the result set.

SQLite

For some strange reason, SQLite has implemented their `ResultSetMetaData` as also being a `ResultSet` which leads to ambiguity when datafying some things when using SQLite. `next.jdbc` currently assumes that if it is asked to `datafy` a `ResultSet` and that object is *also* `ResultSetMetaData`, it will treat it purely as `ResultSetMetaData`, which produces a vector of column metadata as described above. However, there are some results in SQLite's JDBC driver that look like `ResultSetMetaData` but should be treated as plain `ResultSet` objects (which is what other databases' JDBC drivers return).

An example of this is what happens when you try to `datafy` the result of calling `DatabaseMetaData.getTables()`: the JDBC documentation says you get back a `ResultSet` but in SQLite, that is also an instance of `ResultSetMetaData` and so `next.jdbc.datafy` treats it that way instead of as a plain `ResultSet`. You can call `next.jdbc.result-set/datafiable-result-set` directly in this case to get the rows as a hash map (although you won't get the underlying metadata as a bean).

See issue #212 for more details.