RowBuilder and ResultSetBuilder

In Getting Started, it was noted that, by default, `execute!` and `execute-one!` return result sets as (vectors of) hash maps with namespace-qualified keys as-is. If your database naturally produces uppercase column names from the JDBC driver, that's what you'll get. If it produces mixed-case names, that's what you'll get.

> Note: Some databases do not return the table name in the metadata by default. If you run into this, you might try adding `:ResultSetMetaDataOptions "1"` to your db-spec (so it is passed as a property to the JDBC driver when you create connections). If your database supports that, it will perform additional work to try to add table names to the result set metadata. It has been reported that Oracle just plain old does not support table names at all in its JDBC drivers.

The default builder for rows and result sets creates qualified keywords that match whatever case the JDBC driver produces. That builder is `next.jdbc.result-set/as-maps` but there are several options available:

- `as-maps` -- table-qualified keywords as-is, the default, e.g., `:ADDRESS/ID`, `:myTable/firstName`,
- `as-unqualified-maps` -- simple keywords as-is, e.g., `:ID`, `:firstName`,
- `as-lower-maps` -- table-qualified lower-case keywords, e.g., `:address/id`, `:mytable/firstname`,
- `as-unqualified-lower-maps` -- simple lower-case keywords, e.g., `:id`, `:firstname`,
- `as-arrays` -- table-qualified keywords as-is (vector of column names, followed by vectors of row values),
- `as-unqualified-arrays` -- simple keywords as-is,

Clojure's direction (with `clojure.spec` etc), and c) not mess with the data. `as-arrays` is (slightly) faster than `as-maps` since it produces less data (vectors of values instead of vectors of hash maps), but the `lower` options will be slightly slower since they include (conditional) logic to convert strings to lower-case. The `unqualified` options may be slightly faster than their qualified equivalents but **make no attempt to keep column names unique if your SQL joins across multiple tables**.

> Note: This is a deliberate difference from `clojure.java.jdbc` which would make column names unique by appending numeric suffixes. It was always poor practice to rely on `clojure.java.jdbc`'s renaming behavior and it added quite an overhead to result set building, which is why `next.jdbc` does not support it -- use explicit column aliasing in your SQL instead if you want *unqualified* column names!

In addition, the following generic builders can take `:label-fn` and `:qualifier-fn` options to control how the label and qualified are processed. The `lower` variants above are implemented in terms of these, passing a `lower-case` function for both of those options.

- `as-modified-maps` -- table-qualified keywords,
- `as-unqualified-modified-maps` -- simple keywords,
- `as-modified-arrays` -- table-qualified keywords,
- `as-unqualified-modified-arrays` -- simple keywords.

An example builder that naively converts `snake_case` database table/column names to `kebab-case` keywords:

However, a version of `as-kebab-maps` is built-in, as is `as-unqualified-kebab-maps`, which both use the `->kebab-case` function from the camel-snake-kebab library with `as-modified-maps` and `as-unqualified-modified-maps` respectively, so you can just use the built-in `result-set/as-kebab-maps` (or `result-set/as-unqualified-kebab-maps`) builder as a `:builder-fn` option instead of writing your own.

> Note: `next.jdbc/snake-kebab-opts` and `next.jdbc/unqualified-snake-kebab-opts` exist, providing pre-built options hash maps that contain these `:builder-fn` options, as well as appropriate `:table-fn` and `:column-fn` options for the **Friendly SQL Functions** so those are often the most convenient way to enable snake/kebab case conversions with `next.jdbc`.

And finally there are two styles of adapters for the existing builders that let you override the default way that columns are read from result sets. The first style takes a `column-reader` function, which is called with the `ResultSet`, the `ResultSetMetaData`, and the column index, and is expected to read the raw column value from the result set and return it. The result is then passed through `read-column-by-index` (from `ReadableColumn`, which may be implemented directly via protocol extension or via metadata on the result of the `column-reader` function):

- `as-maps-adapter` -- adapts an existing map builder function with a new column reader,
- `as-arrays-adapter` -- adapts an existing array builder function with a new column reader.

The default `column-reader` function behavior would be:

```
(defn default-column-reader
  [^ResultSet rs ^ResultSetMetaData rsmeta ^Integer i]
  (.getObject rs i))
```

```
     {:builder-fn (result-set/as-maps-adapter
                    result-set/as-maps
                    result-set/clob-column-reader)}
```

As of 1.1.569, the second style of adapter relies on `with-column-value` from `RowBuilder` (see below) and allows you to take complete control of the column reading process. This style takes a `column-by-index-fn` function, which is called with the builder itself, the `ResultSet`, and the column index, and is expected to read the raw column value from the result set and perform any and all processing on it, before returning it. The result is added directly to the current row with no further processing.

- `builder-adapter` -- adapts any existing builder function with a new column reading function.

The default `column-by-index-fn` function behavior would be:

```
(defn default-column-by-index-fn
  [builder ^ResultSet rs ^Integer i]
  (result-set/read-column-by-index (.getObject rs i) (:rsmeta builder) i))
```

Because the builder itself is passed in, the vector of processed column names is available as `(:cols builder)` (in addition to the `ResultSetMetaData` as `(:rsmeta builder)`). This allows you to take different actions based on the metadata or the column name, as well as bypassing the `read-column-by-index` call if you wish.

The older `as-*-adapter` functions are now implemented in terms of this `builder-adapter` because `with-column-value` abstracts away *how* the new column's value is added to the row being built.

`ResultSet` as a Clojure data structure:

- `(->row builder)` -- produces a new row (a `(transient {})` by default),
- `(column-count builder)` -- returns the number of columns in each row,
- `(with-column builder row i)` -- given the row so far, fetches column `i` from the current row of the `ResultSet`, converts it to a Clojure value, and adds it to the row (for `as-maps` this is a call to `.getObject`, a call to `read-column-by-index` -- see the `ReadableColumn` protocol below, and a call to `assoc!`),
- `(with-column-value builder row col v)` -- given the row so far, the column name, and the column value, add the column name/value to the row in the appropriate way: this is a low-level utility, intended to be used in builders (or adapters) that want to control more of the value handling process -- in general, `with-column` will be implemented by calling `with-column-value`,
- `(row! builder row)` -- completes the row (a `(persistent! row)` call by default).

`execute!` and `execute-one!` call these functions for each row they need to build. `plan` *may* call these functions if the reducing function causes a row to be materialized.

ResultSet Protocol

This protocol defines three functions and is used whenever `next.jdbc` needs to materialize a result set (multiple rows) from a `ResultSet` as a Clojure data structure:

- `(->rs builder)` -- produces a new result set (a `(transient [])` by default),
- `(with-row builder rs row)` -- given the result set so far and a new row, returns the updated result set (a `(conj! rs row)` call by default),

Result Set Builder Functions

The `as-*` functions described above are all implemented in terms of these protocols. They are passed the `ResultSet` object and the options hash map (as passed into various `next.jdbc` functions). They return an implementation of the protocols that is then used to build rows and the result set. Note that the `ResultSet` passed in is *mutable* and is advanced from row to row by the SQL execution function, so each time `->row` is called, the underlying `ResultSet` object points at each new row in turn. By contrast, `->rs` (which is only called by `execute!`) is invoked *before* the `ResultSet` is advanced to the first row.

The result set builder implementation is also assumed to implement `clojure.lang.ILookup` such that the keys `:cols` and `:rsmeta` are supported and should map to the vector of column names that the builder will produce and the `ResultSetMetaData` object (which can be obtained from the `ResultSet`, if necessary). This is intended to allow `plan` and various builder adapters to access certain information that may be needed for processing results. The default builder implementations (for maps and arrays) are both records with fields `rsmeta` and `cols` (in addition to `rs` -- the `ResultSet` itself). The adapters provided in `next.jdbc.result-set` returned reified implementations that delegate field lookup to the underlying builder implementation.

The options hash map for any `next.jdbc` function can contain a `:builder-fn` key and the value is used as the row/result set builder function. The tests for `next.jdbc.result-set` include a record-based builder function as an example of how you can extend this to satisfy your needs.

> Note: When `next.jdbc` cannot obtain a `ResultSet` object and returns `{:next.jdbc/count N}` instead, the builder function is not applied -- the `:builder-fn` option does not affect the shape of the result.

There is also a convenience function, `datafiable-result-set`, that accepts a `ResultSet` object (and a connectable and an options hash map) and returns a fully realized result set, per the `:builder-fn` option (or `as-maps` if that option is omitted).

The array-based builders warrant special mention:

- When used with `execute!`, the array-based builders will produce a data structure that is a vector of vectors, with the first element being a vector of column names and subsequent elements being vectors of column values in the same corresponding order. The order of column names and values follows the "natural" order from the SQL operation, as determined by the underlying `ResultSet`.
- When used with `execute-one!`, the array-based builders will produce a single vector containing the column values in the "natural" SQL order but you will not get the corresponding column names back.
- When used with `plan`, the array-based builders will cause each abstract row to represent a vector of column values rather than a hash map which limits the operations you can perform on the abstraction to just `Associative` (`get` with a numeric key), `Counted` (`count`), and `Indexed` (`nth`). All other operations will either realize a vector, as if by calling `datafiable-row`, or will fail if the operation does not make sense on a vector (as opposed to a hash map).

`next.jdbc.optional`

This namespace contains variants of the six `as-maps`-style builders above that omit keys from the row hash maps if the corresponding column is `NULL`. This is in keeping with Clojure's views of "optionality" -- that optional elements should simply be omitted -- and is provided as an "opt-in" style of rows and result sets.

ReadableColumn

is part of the `ReadableColumn` protocol that you can extend to handle conversion of arbitrary database-specific types to Clojure values. It is extensible via metadata so the value you return can have metadata specifying the implementation of `read-column-by-index`.

If you need more control over how values are read from the `ResultSet` object, you can use `next.jdbc.result-set/as-maps-adapter` (or `next.jdbc.result-set/as-arrays-adapter`, or the more low-level but more generic `next.jdbc.result-set/builder-adapter`) which takes an existing builder function and a column reading function and returns a new builder function that calls your column reading function (with the `ResultSet` object, the `ResultSetMetaData` object, and the column index -- or the builder itself, the `ResultSet` object, and the column index in the case of `builder-adapter`) instead of calling `.getObject` directly. Note that the `as-*` adapters still call `read-column-by-index` on the value your column reading function returns.

In addition, inside `plan`, as each value is looked up by name in the current state of the `ResultSet` object, the `read-column-by-label` function is called, again passing the column value and the column label (the name used in the SQL to identify that column). This function is also part of the `ReadableColumn` protocol.

The default implementation of this protocol is for these two functions to return `nil` as `nil`, a `Boolean` value as a canonical `true` or `false` value (unfortunately, JDBC drivers cannot be relied on to return unique values here!), and for all other objects to be returned as-is.

`next.jdbc` makes no assumptions beyond `nil` and `Boolean`, but common extensions here could include converting `java.sql.Date` to `java.time.LocalDate` and `java.sql.Timestamp` to `java.time.Instant` for example:

```
(extend-protocol rs/ReadableColumn
  java.sql.Date
  java.sql.Timestamp
  (read-column-by-label [^java.sql.Timestamp v _]
    (.toInstant v))
  (read-column-by-index [^java.sql.Timestamp v _2 _3]
    (.toInstant v)))
```

Remember that a protocol extension will apply to all code running in your application so with the above code **all** timestamp values coming from the database will be converted to `java.time.Instant` for all queries. If you want to control behavior across different calls, consider the adapters described above (`as-maps-adapter` , `as-arrays-adapter` , and `builder-adapter` , and think about using metadata to implement the `rs/ReadableColumn` protocol instead of extending it).

Note that the converse, converting Clojure values to database-specific types is handled by the `SettableParameter` protocol, discussed in the next section (Prepared Statements).

❮ Tips & Tricks Prepared Statements ❯