ChatGPT

HoneySQL（ベクタ形式）と next.jdbc を使った複雑クエリ構築ガイド

概要：HoneySQLの特徴とベクタ形式の利点

HoneySQL は「SQLをClojureのデータ構造で表現する」ライブラリで、クエリを文字列連結ではなくマップやベクタで記述できる [1] 。例えば、以下のように Clojure のマップで基本クエリを定義できます [2] 。

```
(def base-query {:select [:a :b :c]
                 :from   [:foo]
                 :where  [:= :foo.a "baz"]})
;; sql/format を呼ぶと、対応する SQL 文とパラメータのベクタが返る
(sql/format base-query)
;; => ["SELECT a, b, c FROM foo WHERE foo.a = ?" "baz"]
```

このようにクエリがマップやベクタで組み立てられるため、動的・条件的なクエリ生成が容易になります [3] 。複数のWHERE句を条件に応じて追加したり、JOIN句を必要に応じて加えたりといった操作が簡潔に記述でき、コードの可読性・保守性が高まります（後述の例を参照）。また、HoneySQL自体は「SQL文字列を生成するだけ」の純粋ライブラリであり、JDBC接続や実行は行わないため、生成したクエリ文字列を `next.jdbc` などのデータベースライブラリに渡して実行します [4] 。

1. シンプルなSELECTからJOIN・WHEREを追加する例

まずは単純なSELECTクエリをHoneySQLで書いてみましょう。ベクタ形式では `:select` , `:from` , `:where` といったキーに対して値を指定します。

```
(require '[honey.sql :as sql])

(def simple-query {:select [:u.id :u.name]
                   :from   [[:users :u]]})
(sql/format simple-query)
;; => ["SELECT u.id, u.name FROM users AS u"]
```

このクエリにWHERE句を追加するには、 `:where` キーに条件ベクタを指定します。条件は `[:= :u.active true]` のように、先頭要素に演算子を置いたベクタ形式で記述します。

```
(def with-where {:select [:u.id :u.name]
                 :from   [[:users :u]]
                 :where  [:= :u.active true]})
(sql/format with-where)
;; => ["SELECT u.id, u.name FROM users AS u WHERE u.active = ?" true]
```

次にJOIN句を加えてみます。HoneySQLでは `:join`（`INNER JOIN`）、`:left-join`、`:right-join` などで結合を指定します。以下は `users` と `roles` テーブルをユーザIDで結合する例です。

```
(def join-query {:select [:u.id :u.name :r.role_name]
                 :from   [[:users :u]]
                 :join   [[:roles :r] [:= :r.id :u.role_id]]
                 :where  [:= :u.active true]})
(sql/format join-query)
;; => ["SELECT u.id, u.name, r.role_name FROM users AS u INNER JOIN roles AS r ON r.id = u.role_id
WHERE u.active = ?" true]
```

ここで、JOIN句は `:join [[テーブル エイリアス] 条件]` という形で表現します [5]。同様にLEFT JOINやRIGHT JOINを使う場合はキー名を変えるだけで、別テーブルとの結合が可能です。Alias（エイリアス）も `[[:テーブル :alias] ...]` の形で指定でき、上記のように `:from [[:users :u]]` や `:join [[:roles :r] ...]` とします [5]。

このように、HoneySQLではクエリの各句をマップで定義し、それを `(sql/format ...)` で SQL 文字列とパラメータのベクタに変換します。キーの並び順は SQL の順番に従わずともよい（mapなので任意）ですが、`:select`, `:from`, `:join`, `:where` の順に設定すると見通しがよくなります [3] [5]。

2. サブクエリとCTE（WITH句）の分解と再利用

複雑なクエリを扱う場合、サブクエリや共通テーブル式（CTE, WITH句）を使ってクエリを分解し、再利用・可読性を高めることが有効です。HoneySQLでは `:with` キーで CTE を定義できます。`:with` の値には名前とクエリの組をリストで並べます [6]。

例えば、組織情報（`departments`）とユーザ情報（`users`）を扱う場合、まず部門ごとの所属ユーザ数を集計し、その結果を結合してユーザ情報を取得するクエリを考えます。CTEで部門ごとのユーザ数を計算すると、メインクエリがシンプルになります。HoneySQLの例：

```
(def dept-user-count
 ;; CTE定義: 部門ごとのユーザ数を集計
 [[:dept_counts {:select [:d.id :d.name [:count :u.id :user_count]]
                 :from   [[:departments :d]]
                 :join   [[:users :u] [:= :u.dept_id :d.id]]
                 :group-by [:d.id]}]])

(def main-query
 ;; WITH句でCTEを使い、集計結果とJOIN
 (merge {:with   dept-user-count
         :select [:u.id :u.name :dc.user_count]
         :from   [[:users :u]]}
        {:join   [[:dept_counts :dc] [:= :u.dept_id :dc.id]]}))
(sql/format main-query)
;; => ["WITH dept_counts AS (SELECT d.id, d.name, COUNT(u.id) AS user_count FROM departments
AS d INNER JOIN users AS u ON u.dept_id = d.id GROUP BY d.id)
;;    SELECT u.id, u.name, dc.user_count FROM users AS u INNER JOIN dept_counts AS dc ON
u.dept_id = dc.id"]
```

上記のように、まず `:with` でCTE（この例では `dept_counts`）を定義し、次にメインクエリでその `dept_counts` テーブルを `:join` しています。クエリ全体は HoneySQL のマップで組み立てており、最後に `(sql/format main-query)` で SQL に変換します [6]。CTEを使うと同じ計算やフィルタを複数回書く必要がなくなるため、可読性とパフォーマンス上の利点があります。

3. 再帰的 WITH 句（組織階層トラバース）の表現

企業の部門構造やツリー状データを扱う際は、PostgreSQLの再帰的CTE（`WITH RECURSIVE`）を用いて階層構造をたどることができます。HoneySQLでも同様に、`:with-recursive` キーを使って再帰的CTEを定義できます [7]。例えば、部門テーブル `departments` が親子関係を持つツリー構造だとして、最上位の部門から全階層を取得するクエリを作る例です。

```
(def recursive-cte
 ;; 再帰CTEの定義: start を最上位部門とし、その配下を再帰的に探索
 [[:dept_tree {:columns [:id :name :parent_id]
               :select  [:d.id :d.name :d.parent_id]
               :from    [[:departments :d]]
               :where   [:= :d.parent_id nil]}
               ;; 再帰部分: 子を親と結合して追加取得
               {:union-all {:select [:c.id :c.name :c.parent_id]
                            :from   [[:departments :c]]
                            :join   [[:dept_tree :dt] [:= :c.parent_id :dt.id]]}}]])
(def recur-query
 (merge {:with-recursive recursive-cte
         :select         [:*]
         :from           [[:dept_tree :dt]]} {}))
(sql/format recur-query)
;; => ["WITH RECURSIVE dept_tree (id, name, parent_id) AS (
;;      SELECT d.id, d.name, d.parent_id FROM departments AS d WHERE d.parent_id IS NULL
;;      UNION ALL
;;      SELECT c.id, c.name, c.parent_id FROM departments AS c
;;      INNER JOIN dept_tree AS dt ON c.parent_id = dt.id
;;    )
;;    SELECT * FROM dept_tree"]
```

この例では `:with-recursive` を使い、最初に親IDがNULL（ルート部門）を取得し、次に `UNION ALL` で子部門を親部門 `dept_tree` と結合しています。 [7] にあるように、`:with-recursive` は通常の `:with` と同様のルールで指定します。HoneySQLで再帰CTEを表現する場合、上記のように `:union-all` を使って再帰部分を定義します。結果として生成されるSQLはPostgreSQLの `WITH RECURSIVE` 構文となり、階層トラバースが可能です。

4. N+1問題の解説とプリフェッチ対策例

N+1問題とは、例えば「まず全車情報を取得し、続いて1台ごとに車輪情報を個別クエリで取得する」ようなケースで、合計1+N回のクエリを発行してしまうことです [8]。具体的には以下のようなSQL発行が行われ、効率が非常に悪くなります [8]。

```
SELECT * FROM Cars;   -- 1回目
# その後、全てのCarについて以下を繰り返す:
SELECT * FROM Wheel WHERE CarId = ?;  -- Carの台数分（N回）
```

上記のように最初にN件の `Cars` を取得し、その後各 `CarId` で `Wheel` をN回問い合わせると、クエリ回数がN+1 となりパフォーマンス悪化の原因となります [8]。対策として、必要なデータを結合やまとめ取得で一度に取る「プリフェッチ」が有効です [9]。例えば、JOINを使って一度に車と車輪を結合取得したり、取得したいIDリストを使ってIN句でまとめて取得します。

- JOINによる一括取得例：

```
;; ユーザとそのポスト（1対多）を結合して一度に取得
(def users-posts
 {:select [:u.id :u.name :p.title]
  :from   [[:users :u]]
  :left-join [[:posts :p] [:= :p.user_id :u.id]]})
(sql/format users-posts)
;; => ["SELECT u.id, u.name, p.title FROM users AS u LEFT JOIN posts AS p ON p.user_id =
u.id"]
```

このようにJOINでまとめてデータを取得すれば、複数クエリを発行する必要がなくなります。

- IN句によるまとめ取得例：

```
;; まずユーザIDの一覧を取得し、そのIDリストでポストをまとめ取得する
(def user-ids-query {:select [:u.id] :from [[:users :u]]})
;; (ユーザIDを取得してClojureで[1 2 3 ...]の形で集めると仮定)
(def posts-query {:select [:*] :from [:posts]
                  :where  [:in :user_id [1 2 3]]})
(sql/format posts-query)
;; => ["SELECT * FROM posts WHERE user_id IN (?, ?, ?)" 1 2 3]
```

このようにIN句で複数のIDに対する関連データを一度に取得すると、クエリ回数を大幅に削減できます [9]。

これらの手法によって、N+1問題で課題となる大量の往復を回避できます [9]。実際、上記の例のように「`SELECT * FROM Wheel` で全てのwheelを一度に取得すればラウンドトリップ数は2回になる（Cars と Wheel各1回）」と解説されています [9]。HoneySQLを使えば上記のJOIN句やIN句のSQLもデータ構造で表現できるため、ORMでありがちなN+1問題も手動で対策しやすくなります。

5. next.jdbcとの統合：クエリ構築→実行→結果処理パターン

HoneySQLで組み立てたクエリを実行するには、JDBCライブラリ（ここでは `next.jdbc`）に渡します。基本的な流れは以下の通りです。

1. **データソースの設定**
   DB接続情報を定義し、`jdbc/get-datasource` で `DataSource` を作成します（以下はPostgreSQLの

例です）。

```
(require '[next.jdbc :as jdbc])
(def db-config {:dbtype "postgresql" :host "localhost" :dbname "mydb"
                :user "user" :password "pass"})
(def ds (jdbc/get-datasource db-config))
```

2. **HoneySQLでクエリを組み立てる**
   先述の方法でクエリマップを作り、`sql/format` でSQL文字列＋パラメータのベクタを得ます。

```
(require '[honey.sql :as sql])
(def query {:select [:u.id :u.name] :from [[:users :u]] :where [:= :u.active true]})
(def sql-and-params (sql/format query))
;; sql-and-params => ["SELECT u.id, u.name FROM users AS u WHERE u.active = ?" true]
```

3. **クエリの実行**
   生成したベクタを `jdbc/execute!` に渡して実行します。`execute!` は結果をキーワードマップのベクタで返します。

```
(def results (jdbc/execute! ds sql-and-params))
;; => [{:users/id 1, :users/name "alice"} {:users/id 2, :users/name "bob"} ...]
```

   実際の実行例を見ると、HoneySQLでマップを作り `sql/format` したものを `execute!` に渡しています [10]。実行結果は名前空間付きキーワード（テーブル名/カラム名）をキーにしたマップのリストで返ってきます [10]。

4. **結果の処理**
   `jdbc/execute!` の返り値はベクタなので、必要に応じて `first` や `map` で取り出し処理します。例えば1件取得時は `(first results)`、複数件取得時は `map` でループ処理します。

HoneySQLとnext.jdbcの連携はこのようにシンプルで、生成したクエリデータ構造をそのままDB実行に使えるのが特徴です [4] [10]。生成SQLにパラメータはプレースホルダ（`?`）で安全にバインドされ、SQLインジェクションのリスクも軽減されます。

6. コンポーネントベースのクエリ生成

HoneySQLではクエリの各部分が普通のClojureデータなので、再利用可能な「コンポーネント」として分割し、合成することが容易です。例えば、ベースクエリ、フィルタ条件、JOIN句などを別々のマップで定義し、必要に応じて `merge` や `merge-with` で結合できます。

```
(def base-query {:select [:u.id :u.name]
                 :from   [[:users :u]]})
(def active-filter {:where [:= :u.active true]})
(def org-join {:left-join [[:organizations :o] [:= :o.id :u.org_id]]})

;; コンポーネントマップを結合して最終クエリを構築
(def final-query (merge base-query active-filter org-join))
```

```
(sql/format final-query)
;; => ["SELECT u.id, u.name FROM users AS u LEFT JOIN organizations AS o ON o.id = u.org_id WHERE u.active = ?" true]
```

ここでは `merge` で単純なキーの結合を行っていますが、WHERE句やJOIN句を複数まとめたい場合は `merge-with into` や HoneySQL提供のヘルパー関数を使うこともできます。例えば `merge-where` ヘルパーを使えば、複数のWHERE条件を論理ANDでまとめられます [11]。

```
(require '[honey.sql.helpers :refer [merge-where]])
(def partial-query {:where [:= :u.age 30]})
(def more-cond     [:= :u.country "JP"])
(def merged (merge-where partial-query more-cond))
;; merged => {:where [:and [:= :u.age 30] [:= :u.country "JP"]]}
```

このように、クエリの断片（フィルタや結合設定）をマップで定義し、関数で合成できるのがHoneySQLの利点です [3] [11]。また、`cond->` や普通の `if` 文と組み合わせて条件的にクエリ断片を追加することも可能です [3]。これにより複雑なクエリを可読性高く分解・再利用し、必要なときに結合していく「コンポーネントベース」の設計が実現できます。

まとめ

以上のように、HoneySQL（ベクタ形式）と next.jdbc を組み合わせることで、複雑なPostgreSQLクエリを可読性高く構築・実行できます。HoneySQLのデータ構造ベースのDSLを使えば、クエリをマップで組み立て、動的条件やJOIN、再帰CTE、サブクエリを自然に表現できます [2] [6]。また、N+1問題への対策としてJOINやIN句を活用したプリフェッチも簡潔に記述でき、パフォーマンスを向上できます [8] [9]。生成したクエリは次世代JDBCライブラリのnext.jdbcでそのまま実行でき、結果も扱いやすいマップ形式で取得できます [4] [10]。こうしてクエリを小さな断片（コンポーネント）として定義・結合することで、大規模・複雑なSQLを整然と管理するベストプラクティスが実現します。

参考資料: HoneySQLドキュメント [1] [6]、StackOverflowの解説 [8] [11] など。

---

[1] [2] [3] [4] GitHub - seancorfield/honeysql: Turn Clojure data structures into SQL
https://github.com/seancorfield/honeysql

[5] SQL Clause Reference — com.github.seancorfield/honeysql 2.7.1340
https://cljdoc.org/d/com.github.seancorfield/honeysql/2.7.1340/doc/getting-started/sql-clause-reference

[6] [7] SQL Clause Reference — seancorfield/honeysql 2.0.0-rc2
https://cljdoc.org/d/seancorfield/honeysql/2.0.0-rc2/doc/getting-started/sql-clause-reference

[8] [9] database - What is the "N+1 selects problem" in ORM (Object-Relational Mapping)? - Stack Overflow
https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping

[10] PostgreSQL with Clojure and HoneySQL | by Santhosh Krishnamoorthy | Medium
https://santhoshkris.medium.com/postgres-with-clojure-and-honeysql-cb666a28d803

[11] honeysql.helpers — honeysql 1.0.461
https://cljdoc.org/d/honeysql/honeysql/1.0.461/api/honeysql.helpers