PGQA Mode

GNU Emacs mode to parse, format and analyze SQL queries

Antonín Houska

1 Introduction

PGQA (PostgreSQL Query Analyzer) is a major mode of GNU Emacs editor, designed to parse, format and analyze SQL queries for PostgreSQL database server.

Besides providing the user with particular functionality (accessible via menu and key sequences), the project aims to offer low-level functions to search in the query tree and to modify it.

2 Getting Started

First, make sure you have GNU Emacs 25 or later installed.

Then download the latest version of PGQA from `http://www.melesmeles.cz/pgqa` and use *`M-x package-install-file`* sequence to install it. That is, press `ESC` followed by `x`, type `package-install-file` command name, press `RET` and enter path to the PGQA package.

Once you open a file having `sql` suffix, major mode of the containing buffer should automatically become the PGQA mode. If the file has different suffix, you can use *`M-x pgqa-mode`* sequence to activate the mode. Once the mode has been activated, "PGQA" string should appear in the mode line.

2.1 Installation for Developers

If you want to hack PGQA, you'll probably prefer the following installation procedure:

1. Clone the Git repository
2. To ensure that the code gets loaded, add the repository directory to the value of `EMACSLOADPATH`. For example, run the following command (and add it to your `.bashrc` file so that you don't have to run it in next time again):

   ```
   export EMACSLOADPATH=:~/repos/pgqa/
   ```

   (The initial colon ensures that the existing value of *load-path* variable is not discarded.)
3. Then add the following lines to the init file of GNU Emacs (`.gnu-emacs` in your home directory):

   ```
   ;; Enable autoloading of the PGQA mode.
   (autoload 'pgqa-mode "pgqa" "PGQA major mode function." t)

   ;; Make sure *.sql file suffix activates the PGQA mode automatically.
   (add-to-list 'auto-mode-alist (cons "\\.sql\\'" 'pgqa-mode))

   ;; pgqa-format-query does not require the pgqa-mode. (Syntax
   ;; highlighting is not active w/o the pgqa-mode.)
   (autoload 'pgqa-format-query "pgqa" "Format SQL query." t)
   ```
4. Finally restart the GNU Emacs editor.

3 Customization

C-c + key sequence runs `pgqa-customize` command which opens a window containing all customization settings of the PGQA mode. Particular settings are explained in the related sections.

4 Query Parsing

4.1 Interactive Mode

If the PGQA mode is active, `C-c >` key sequence runs command `pgqa-parse`, which considers the current buffer to contain a single SQL query. The command parses the query string and puts the internal format (tree) to `pgqa-query-tree` buffer-local variable. The tree consists of nodes that represent query parts such as tables, joins, expressions, etc. The tree is displayed in `*pgqa-query-tree-text*` buffer.

In addition, the `pgqa-parse` function sets highlights some nodes of the query by setting their `font-lock-face` text property, see Section 4.3 [Parsing Customization], page 4.

Problems that PGQA finds in the query during parsing (or during subsequent analysis) appear in the `*pgqa-log*` buffer. Each error / warning message appears there as a hyperlink so you can navigate to the related part of the query by clicking on the message text.

If you only want to parse part of the buffer (the current buffer may contain some other text besides the SQL query that PGQA parser does not recognize, e.g. constructs of PL/pgSQL language), or if the buffer contains multiple queries, user can mark the query (i.e. put it into "region"). Thus the `pgqa-parse` command only processes the region contents.

The region is remembered, so the there's no need to select the query again before the next run of `pgqa-parse`.

4.2 Batch Mode

PGQA can parse query in batch mode. This requires `EMACSLOADPATH` environment variable to be adjusted as described in see Chapter 2 [Getting Started], page 2.

To get file parsed in batch mode, run GNU Emacs this way (query.sql is the file containing your SQL query):

```
emacs -batch --insert query.sql -l pgqa.el -f pgqa-parse-query-batch
```

The (raw) query will be sent to the standard output.

4.3 Customization

This section summarizes the subset of customization settings (see Chapter 3 [Customization], page 3) that affect SQL query parsing.

`pgqa-query-tree-print-region` tells whether the machine-readable output of the parser should contain region information for each node, i.e. where particular node is located in the source buffer (i.e. the buffer that contains the query in the textual form). The default value is `t`, i.e. the region info is displayed.

Note that the region info might not be accurate if the buffer specified by `pgqa-query-tree-buffer` was refreshed by `pgqa-format-query` function, as opposed to `pgqa-parse`. The point is that `pgqa-format-query` displays the query tree before the actual formatting starts, and the formatting essentially changes node positions. If you're interested in the region info of the formatted query, run `pgqa-parse` on it.

`pgqa-operator-face` contains the face to highlight SQL operators. This setting also applies to Section 5.3 [Formatting Customization], page 6.

`pgqa-func-call-face` contains the face to highlight SQL function calls. This setting also applies to Section 5.3 [Formatting Customization], page 6.

5 Query Formatting

5.1 Interactive Mode

If the PGQA mode is active, `C-c <` key sequence runs command `pgqa-format-query`, which considers the current buffer to contain a single SQL query. The command parses the query string, turns the internal format back into text and replaces the original query with it.

The command tries not to change position of the query within the buffer. In particular, the formatted query starts on the same line as the original did. If the first character of the query does not start at the beginning of a line, indentation of the whole query is adjusted so that the number of spaces in front of each line of the query is whole multiple of `tab-width`. (Spaces are added or removed so that the closest TAB position is reached.)

If the command is called with a prefix argument N, then N is considered the desired TAB position and no estimate is calculated.

If the region is used and the first line is preceded by at least one non-whitespace character, then the indentation is still estimated (or accepted as a prefix argument), but it's not applied to the first line. The idea is that user should know why the non-white character is there.

As for query selection (region), `pgqa-format-query` behaves in the same way as `pgqa-parse`, i.e. the last region is used for the next executions until user performs a new selection.

`pgqa-format-query` can (as long as autoloading is configured, see Chapter 2 [Getting Started], page 2) be used even in buffer whose major mode is not PGQA, but no syntax highlighting is active in that case. Also no key sequence is automatically bound to the command.

`fill-column` variable is honored during the formatting.

5.2 Batch Mode

PGQA can be used to format queries in batch mode. This requires `EMACSLOADPATH` environment variable to be adjusted as described in see Chapter 2 [Getting Started], page 2.

In order to get a query formatted, run GNU Emacs this way (query.sql is the file containing your SQL query):

```
emacs -batch --insert query.sql -l pgqa.el -f pgqa-format-query-batch
```

The formatted query will be sent to the standard output.

5.3 Customization

This section summarizes the subset of customization settings (see Chapter 3 [Customization], page 3) that affect SQL query formatting.

`pgqa-multiline-query` setting can effectively disable formatting if it's value is `nil`. In that case line is only broken if the column of the next character is greater than the value of `fill-column` setting.

For example, if `fill-column` is set to 40 (just to demonstrate how lines are broken, without using too complex query), `pgqa-format-query` will produce this:

```
SELECT p.name, l.name, r.version, r.date
```

```
FROM projects AS p JOIN licenses AS l ON
l.id = p.license_id JOIN releases AS r
ON r.project_id = p.id WHERE r.date <=
'2012-06-30' AND p.name LIKE '%GNU%'
ORDER BY r.date, p.name;
```

(Note that all the following formatting settings must be set to `nil` in this case, otherwise the formatting will result in error message.)

If you set `pgqa-multiline-query` option, each query "clause keyword" (`SELECT`, `FROM`, etc.) of the formatted query will start on a new line:

```
SELECT      p.name, l.name, r.version,
            r.date
FROM        projects AS p JOIN licenses
            AS l ON l.id = p.license_id
            JOIN releases AS r ON
            r.project_id = p.id
WHERE       r.date <= '2012-06-30' AND
            p.name LIKE '%GNU%'
ORDER BY    r.date, p.name;
```

`pgqa-clause-newline` can be set in addition to ensure that the actual "top-level clause" will start on a new line. The clause will be given extra indentation relative to the "clause keyword". For example, if the value of `tab-width` setting is equal to 4, the query will looke like this:

```
SELECT
    p.name, l.name, r.version, r.date
FROM
    projects AS p JOIN licenses AS l ON l.id =
    p.license_id JOIN releases AS r ON
    r.project_id = p.id
WHERE
    r.date <= '2012-06-30' AND p.name LIKE '%GNU%'
ORDER BY
    r.date, p.name;
```

`pgqa-clause-newline` requires `pgqa-multiline-query` to be set.

Furthermore, `pgqa-clause-item-newline` setting ensures that comma in the “top-level” clause is always followed by a new line:

```
SELECT
    p.name,
    l.name,
    r.version,
    r.date
FROM
    projects AS p JOIN licenses AS l ON l.id =
    p.license_id JOIN releases AS r ON
    r.project_id = p.id
WHERE
    r.date <= '2012-06-30' AND p.name LIKE '%GNU%'
ORDER BY
    r.date,
    p.name;
```

`pgqa-clause-item-newline` requires `pgqa-clause-newline` to be set.

`pgqa-multiline-join` setting ensures that `JOIN` keyword is always printed on a new line, following the appropriate indentation:

```
SELECT
    p.name,
    l.name,
    r.version,
    r.date
FROM
    projects AS p
    JOIN licenses AS l ON l.id = p.license_id
    JOIN releases AS r ON r.project_id = p.id
WHERE
    r.date <= '2012-06-30' AND p.name LIKE '%GNU%'
ORDER BY
    r.date,
    p.name;
```

`pgqa-multiline-join` requires `pgqa-multiline-query` to be set.

If `pgqa-join-newline` is enabled, line delimiter and indentation are also printed out in front of the right side of the join:

```
SELECT
    p.name,
    l.name,
    r.version,
    r.date
FROM
    projects AS p
    JOIN
    licenses AS l ON l.id = p.license_id
    JOIN
    releases AS r ON r.project_id = p.id
WHERE
    r.date <= '2012-06-30' AND p.name LIKE '%GNU%'
ORDER BY
    r.date,
    p.name;
```

`pgqa-join-newline` requires `pgqa-multiline-join` to be set.

If `pgqa-clause-keyword-right` is enabled, the clause keywords are right-aligned:

```
  SELECT p.name, l.name, r.version, r.date
    FROM projects AS p
         JOIN
         licenses AS l ON l.id = p.license_id
         JOIN
         releases AS r ON r.project_id = p.id
   WHERE r.date <= '2012-06-30' AND p.name LIKE '%GNU%'
ORDER BY r.date, p.name
```

`pgqa-clause-keyword-right` requires `pgqa-multiline-query` to be set.

If `pgqa-multiline-case` is enabled, each branch of `CASE` expression is printed on a new line:

```
SELECT
        p.name,
        CASE
                WHEN l.name = 'GPL' THEN 'copyleft'
                WHEN l.name IN ('BSD', 'MIT') THEN 'permissive'
                ELSE 'unrecognized'
        END
FROM
        projects AS p
        JOIN licenses AS l ON l.id = p.license_id;
```

`pgqa-multiline-case` requires `pgqa-multiline-join` to be set.

If `pgqa-multiline-case-branch` is enabled, each branch of `CASE` expression is broken into 2 lines. In addition, the branches are separated by an empty line.

```
SELECT
        p.name,
        CASE
                WHEN l.name = 'GPL'
                THEN 'copyleft'

                WHEN l.name IN ('BSD', 'MIT')
                THEN 'permissive'

                ELSE 'unrecognized'
        END
FROM
        projects AS p
        JOIN licenses AS l ON l.id = p.license_id;
```

`pgqa-multiline-case-branch` requires `pgqa-multiline-case` to be set.

`pgqa-multiline-operator` setting can help you understand complex expressions. If this is set, operator expressions are printed out in structured way — exactly the way PGQA understands them:

```
SELECT
    p.name,
    l.name,
    r.version,
    r.date
FROM
    projects AS p
    JOIN
    licenses AS l ON
            l.id
        =
            p.license_id
    JOIN
    releases AS r ON
            r.project_id
        =
            p.id
WHERE
            r.date
        <=
            '2012-06-30'
    AND
            p.name
        LIKE
            '%GNU%'
ORDER BY
    r.date,
    p.name;
```

`pgqa-multiline-operator` requires both `pgqa-multiline-join` and `pgqa-multiline-case-branch` to be set.

`pgqa-print-as` setting determines whether expression or table alias will be denoted by the `AS` keyword.

6 Sending SQL Queries to Database

If region has been marked and if it contains an SQL query, `pgqa-send-region` function can be used to send it to (PostgreSQL) database server. The output will be written (appended) to buffer whose name is in the `pgqa-psql-output` variable. (If the region contents is not a valid SQL query, it'll be sent but the appropriate error message will appear in the output buffer instead of query result.)

`C-c )` key sequence can be used to invoke the `pgqa-send-region` function.

6.1 Customization

This section summarizes the subset of customization settings (see Chapter 3 [Customization], page 3) that affect SQL query parsing.

`pgqa-host` variable specifies network name of the machine on which PostgreSQL server is running. The default value is `localhost`.

`pgqa-port` variable specifies TCP port on which PostgreSQL server is running. The default value is `5432`.

`pgqa-database` variable specifies name of the database to which the `pgqa-send-region` function will send queries. The default value is `postgres`.

`pgqa-role`variable specifies the role (user) that the `pgqa-send-region` function uses to connect to the database. The default value is `postgres`.

`pgqq-psql-path` variable specifies path to the `psql` client appication that `pgqa-send-region` function uses to connect to database. The default value is `psql`.

`pgqa-query-tree-buffer` variable specifies name of the buffer that will receive response of the PostgreSQL database server to the query. The default value is `*pgqa-psql-output*`.

7 Treating Query as SQL String

Sometimes it's useful to have application (typically PL/pgSQL function, see Executing Dynamic Commands (`https://www.postgresql.org/docs/10/static/plpgsql-statements.html#PLPGSQL-STATEMENTS-EXECUTING-DYN`)) construct the SQL query in a form of SQL string. The PGQA mode supports this approach by providing the following functions.

`pgqa-query-to-string` converts the query in the current region to an SQL string. Formatting of the query is controlled by the same customization as if it was performed by `pgqa-format-query`. Likewise, if a prefix argument is passed, the function interprets it as the desired indentation of the resulting query.

The `pgqa-query-to-string` function can be executed by *`C-c {`* key sequence.

`pgqa-create-insertion` turns text in the current region into an "SQL insertion". The next call of `pgqa-query-to-string` will not convert this part into the SQL string. Instead, the conversion to the SQL string will stop just in front of the insertion and continue after that. The insertion will be separated from the preceding and the following SQL strings by SQL concatenation operator (`||`).

For example, if `foo` in the following query is marked

```
SELECT * FROM foo WHERE i = 1
```

and `pgqa-create-insertion` is called, user is prompted for an "Insertion key". Assuming that PL/pgSQL variable `v_table` will be used to provide the table name, `v_table` seems like a suitable key. Once the insertion is there, `pgqa-query-to-string` produces this query string:

```
'SELECT * FROM ' || v_table || ' WHERE i = 1'
```

`pgqa-string-to-query` function does the oposite to `pgqa-query-to-string`, i.e. it converts SQL string to the actual query. If the query string contains unquoted parts separated from the rest by SQL concatenation operator (`||`), these are treated as SQL insertions.

PGQA remembers the region used as input by `pgqa-string-to-query`. Thus, after you've modified the query (which might include addition of new insertions), you do not need to mark the query again before calling `pgqa-query-to-string`.

The `pgqa-string-to-query` function can be executed by *`C-c }`* key sequence.