FOLIO Report Development Standards

This page provides information about standards and guidelines for queries to the CUL FOLIO Canned Reports Directory.
Some of these standards are adapted from the Open Library Foundation's FOLIO Analytics report repository.

Easy Access to Report Filters

Most queries include parameters that allow those running the reports to adapt the result set to their needs. For instance, if you are interested in a report showing loan activity for one library location, you can include that location in the parameters to filter your results.

Filter parameters are include at the top of the query in the WITH statement. For example,

```
WITH parameters AS (
SELECT
/* enter invoice payment start date and end date in YYYY-MM-DD format */
'2021-07-01' :: DATE AS start_date,
'2022-06-30' :: DATE AS end_date,
/* enter fund group name as 'Central, Humanities, Area Studies, Rare & Distinctive, Law, Sciences, or Social
Sciences' */
''::VARCHAR AS fund_group_filter,
/* enter one or more fund codes separated by commas, as in 'math, music' */
''::VARCHAR AS fund_code_filter,
/* enter one or more fund types separated by commas, as in 'restricted, unrestricted' */
''::VARCHAR AS fund_type_filter,
),
```

Naming Queries

All canned reports developed for the CUL FOLIO Analytics repository are coded with "CR"
followed by a 3-digit number and a short title with words separated by underscores,
as shown in these examples:

* CR104 claims_returned
* CR123 open_orders
* CR130 fund_expenditures_by_po_line

Including a README.md file

Each canned query must be submitted with an associated README.md file. The README.md file documents the purpose of the query, lists the main tables used in the query, and provides instructions for using the query.

SQL Style

- spacing/indentation
    - four spaces for indents

```
include example
```

    - when listing more than 3 elements, put each on a new line (including first)

```
SELECT
    sp.name AS service_point_name,
    m.name AS material_type,
    i.barcode AS item_barcode,
    ...
```

- alternate: keep first element on same line, then indent remaining elements to the location of the first element

```
SELECT sp.name AS service_point_name,
       m.name AS material_type,
       i.barcode AS item_barcode,
       ...
```

- if you have parallel expressions, you may wish to use spacing to align similar elements

```
loan_date BETWEEN (SELECT start_date FROM parameters) AND
                  (SELECT end_date FROM parameters)
```

- keywords
  - write keywords in all caps. Examples:
    - `SELECT`
    - `'2019-01-01' :: DATE`
  - always use AS for aliasing (columns, subqueries, tables, etc.)
- blank lines
  - no blank lines
- punctuation
  - `,` at end of line
  - `(` at end of line
  - `)` at beginning of line, lined up with keyword from line with (
- type conversion
  - always use `' :: '` followed by data type in upper case (e.g., `VARCHAR`, `DATE`)
- comments
  - `/* ... */` for multi-line comments
  - `--` for single line comments
- file name
  - use underscores instead of dashes
- selecting fields
  - Do not use `SELECT *`. List all fields explicitly.
  - (for joins, can join on whole table, don't need a subquery to limit the right table in the join)

Structuring a Query

1. header comment section
   - last edited date? current as of?
   - fields requested in output, in requested order
   - any filters?
   - aggregated or not? (how?)
   - any other context necessary to understand query
   - warning if query might result in more than 1 million rows (Excel)?
   - have this header as a template in the documentation
2. parameters (using `WITH` statement)
   - place parameters at beginning of file to make it easier for people to modify
   - always use name "parameters"
   - avoid using parameter field names that duplicate LDP field names, if possible
   - set default parameter values in a way that should guarantee the query will return some results, both for testing and for reassuring query users
     - if filtering by a date range, use a default date range that is very large (10+ years), even if this query will typically be used for a single year
     - if filtering by value in a particular field (e.g., a particular service point), consider using the most common value
3. additional `WITH` statements to label subqueries (see services_usage query for example) - optional
4. primary query - example of basic structure
   a. SELECT
   b. FROM
   c. WHERE
   d. ORDER BY
5. (add link to good PostgreSQL dictionary)

Details on Specific Strategies

- `WITH` statements
  - can use `WITH` to create temporary tables at the beginning of the query that then get used later

    - last `WITH` statement goes straight into primary `SELECT` statement for query, do not need a comma after last `WITH` statement
    - while in `WITH` statements you can specify the column names before the `SELECT` statement, the code is more readable if you continue to alias the columns with AS instead the `SELECT` statement (see services_usage query)
    - modern SQL article on WITH statements
    - using WITH statements to create Literate SQL
- Catching empty string & null values
    - if you are just selecting a column that may have a null value, you don't need to do anything special
    - if you are transforming the column in some way, like using it in a mathematical calculation or extracting some part of the value you need to test for a null value or empty string
    - one way might be `COALESCE`, which allows you to specify a default value if the result is null
- Picking which table to select from first
    - when writing a query, it's important to think through which table you list first in the `SELECT` statement because of the joins that will build on it
    - start with the table that best represents what you want on each line of the results table
        - for example, if you ultimately want a list of loans, start with loans table
- `LEFT JOIN` vs. `INNER JOIN`
    - in general, using `LEFT JOIN` makes sure you don't accidentally lose the items you're most interested
        - for example, if you're interested in loans and also want to see the demographics of the users making the loan, you can use `LEFT JOIN` to keep all loans even if you don't know the user's demographics
    - if you are filtering a table based on a field in a secondary table, you may instead want to use INNER JOIN to make sure to exclude records that don't have the required value
- `BETWEEN`
    - note that using `BETWEEN` for dates is risky because it only includes records up to midnight of the end date (essentially, the end of the day before, but it will include items exactly at midnight of the end date)
    - if you do use `BETWEEN`, try to educate people about its behavior in comments and set default values that make sense for the behavior (e.g., the first day of one year and the first day of the following year, instead of the last day of the year)
    - if you do not want to risk including values from midnight of the end date, you can use `>= start_date` and `< end_date` instead of `BETWEEN`. This is like using `BETWEEN` except that you use < instead of <=. You still have to use an end date that will not be included in the date range (i.e., the day after the last day you want included).
    - stack overflow question on querying between date ranges
    - (add how to use INTEGER with HRIDs)
- DRY - Don't Repeat Yourself
    - as with any programming, the more repetition you have in your query, the more likely you are to forget to update something or make a mistake the second time around
    - try to find a way to reuse parts of your query creatively, either with parameters or `WITH` statements