Introduction to dbtools

Introduction to dbtools

dbtools is a library used to query AWS Athena databases from R on the Ministry of Justice's Analytical Platform. It uses the Python library pydbtools and inherits much of its functionality, including creating and querying temporary tables and injecting SQL queries with template arguments.

```
library(dbtools)
```

Reading SQL queries

The read_sql_query function is used to obtain R dataframes from SQL queries sent to Athena.

```
read_sql_query("select * from aws_example_dbtools.employees limit 5")
#> # A tibble: 5 x 6
#>   employee_id sex   forename surname  department_id manager_id
#>         <int> <chr> <chr>    <chr>            <dbl>      <dbl>
#> 1           1 M     Dexter   Mitchell             1         17
#> 2           2 F     Summer   Bennett              1         17
#> 3           3 M     Pip      Carter               1         17
#> 4           4 F     Bella    Long                 1         17
#> 5           5 F     Lexie    Perry               NA         17
```

If a tibble is preferred the read_sql function is provided

```
read_sql("select * from aws_example_dbtools.department limit 5",
         return_df_as="tibble")
#> # A tibble: 5 x 2
#>   department_id department_name
#>           <int> <chr>
#> 1             1 Sales
#> 2             2 Admin
#> 3             3 Management
#> 4             4 Technical
#> 5             5 Maintenance
```

or for a data.table

```
read_sql("select * from aws_example_dbtools.sales limit 5",
         return_df_as="data.table")
#>    employee_id qtr  sales
#> 1:           1   1 768.17
#> 2:           2   1 391.98
#> 3:           3   1 406.36
#> 4:           4   1 816.25
#> 5:           5   1 437.05
```

Creating temporary SQL tables

The `create_temp_table` function allows you to create tables which can be referred to in subsequent queries from the `__temp__` database. For example, to create a table showing total sales per employee from the tables above create a temporary total sales table.

```
sql <- "
SELECT employee_id, sum(sales) as total_sales
FROM aws_example_dbtools.sales
GROUP BY employee_id
"
create_temp_table(sql, table_name="total_sales")
```

Then create a table of employees from the sales department.

```
sql <- "
SELECT e.employee_id, e.forename, e.surname, d.department_name
FROM aws_example_dbtools.employees AS e
LEFT JOIN aws_example_dbtools.department AS d
ON e.department_id = d.department_id
WHERE e.department_id = 1
"
create_temp_table(sql, table_name="sales_employees")
```

The two temporary tables can then be joined to provide the final table.

```
sql <- "
SELECT se.*, ts.total_sales
FROM __temp__.sales_employees AS se
INNER JOIN __temp__.total_sales AS ts
ON se.employee_id = ts.employee_id
"
read_sql_query(sql)
#> # A tibble: 41 x 5
#>    employee_id forename surname   department_name total_sales
#>          <int> <chr>    <chr>     <chr>                 <dbl>
#>  1           1 Dexter   Mitchell  Sales                 2912.
#>  2           2 Summer   Bennett   Sales                 1786.
#>  3           3 Pip      Carter    Sales                  2591.
#>  4           4 Bella    Long      Sales                  2997.
#>  5           6 Robert   Roberts   Sales                  2208.
#>  6           7 Iris     Alexander Sales                  2465.
#>  7           9 Evan     Carter    Sales                  2280.
#>  8          10 Lauren   Powell    Sales                  1936.
#>  9          11 Alice    James     Sales                  3093.
#> 10          12 Owen     Scott     Sales                  2286.
#> # ... with 31 more rows
```

SQL templating

Sometimes you will want to run similar SQL queries which differ only by, for example, table or column names. In these cases SQL templates can be created to SQL queries populated by templated variables, using Jinja2 templating (https://jinja2docs.readthedocs.io/en/stable/index.html). For example,

```
sql_template = "select * from {{ db_name }}.{{ table }} limit 10"
sql <- render_sql_template(sql_template,
                           list(db_name="aws_example_dbtools",
                                table="department"))
sql
#> [1] "select * from aws_example_dbtools.department limit 10"
```

The rendered SQL can then be used to query Athena as usual.

```
read_sql_query(sql)
#> # A tibble: 6 x 2
#>   department_id department_name
#>           <int> <chr>
#> 1             1 Sales
#> 2             2 Admin
#> 3             3 Management
#> 4             4 Technical
#> 5             5 Maintenance
#> 6             6 HR
```

The same template can be used to read a different table.

```
sql <- render_sql_template(sql_template,
                           list(db_name="aws_example_dbtools",
                                table="sales"))
read_sql_query(sql)
#> # A tibble: 10 x 3
#>    employee_id   qtr sales
#>          <int> <int> <dbl>
#>  1           1     1  768.
#>  2           2     1  392.
#>  3           3     1  406.
#>  4           4     1  816.
#>  5           5     1  437.
#>  6           6     1  385.
#>  7           7     1  821.
#>  8           8     1  398.
#>  9           9     1  899.
#> 10          10     1  439.
```

Perhaps more usefully we can use SQL templates saved as a file, which means we can make use of our editors' and IDEs' SQL capabilities.

```
cat("SELECT * FROM {{ db_name }}.{{ table_name }}", file="tempfile.sql")

sql <- get_sql_from_file("tempfile.sql",
                         jinja_args=list(db_name="aws_example_dbtools",
                                         table_name="department"))
read_sql_query(sql)
#> # A tibble: 6 x 2
#>   department_id department_name
#>           <int> <chr>
#> 1             1 Sales
#> 2             2 Admin
#> 3             3 Management
#> 4             4 Technical
#> 5             5 Maintenance
#> 6             6 HR
```

Advanced usage

Creating and maintaining database tables in Athena

In this section we will create a new database from our existing database in Athena. Use the `start_query_execution_and_wait` function to run the SQL creating the database.

```
sql <- "
CREATE DATABASE IF NOT EXISTS new_db_dbtools
COMMENT 'Example of running queries and insert into'
LOCATION 's3://alpha-everyone/dbtools/new_db/'
"

response <- start_query_execution_and_wait(sql)
response$Status$State
#> [1] "SUCCEEDED"
```

Create a derived table in the new database with a CTAS query that both generates the output into S3 and creates the schema of the table. Note that this only inserts the data from quarters 1 and 2.

```
sql <- "
CREATE TABLE new_db_dbtools.sales_report WITH
(
    external_location='s3://alpha-everyone/dbtools/new_db/sales_report'
) AS
SELECT qtr as sales_quarter, sum(sales) AS total_sales
FROM aws_example_dbtools.sales
WHERE qtr IN (1,2)
GROUP BY qtr
"

response <- start_query_execution_and_wait(sql)
response$Status$State
#> [1] "SUCCEEDED"
```

We can now use an insert into query to add the data from quarters 3 and 4 as the schema has already been created.

```
sql <- "
INSERT INTO new_db_dbtools.sales_report
SELECT qtr as sales_quarter, sum(sales) AS total_sales
FROM aws_example_dbtools.sales
WHERE qtr IN (3,4)
GROUP BY qtr
"

response <- start_query_execution_and_wait(sql)
read_sql_query("select * from new_db_dbtools.sales_report")
#> # A tibble: 4 x 2
#>   sales_quarter total_sales
#>           <int>       <dbl>
#> 1             3      26419.
#> 2             4      27559.
#> 3             1      28168.
#> 4             2      30697.
```

Creating a table with partitions

Do the same as before but partition the data based on when the report was run. This can make queries more efficient as filtering on the partition columns reduces the amount of data scanned, plus makes incrementally adding data easier.

```
sql <- "
CREATE TABLE new_db_dbtools.daily_sales_report WITH
(
    external_location='s3://alpha-everyone/dbtools/new_db/daily_sales_report',
    partitioned_by = ARRAY['report_date']
) AS
SELECT qtr as sales_quarter, sum(sales) AS total_sales,
date '2021-01-01' AS report_date
FROM aws_example_dbtools.sales
GROUP BY qtr, date '2021-01-01'
"

response <- start_query_execution_and_wait(sql)
response$Status$State
#> [1] "SUCCEEDED"
```

Then, simulating a source database that is updated daily, add more partitioned data.

```
sql <- "
INSERT INTO new_db_dbtools.daily_sales_report
SELECT qtr as sales_quarter, sum(sales) AS total_sales,
date '2021-01-02' AS report_date
FROM aws_example_dbtools.sales
GROUP BY qtr, date '2021-01-02'
"

response <- start_query_execution_and_wait(sql)
read_sql_query("select * from new_db_dbtools.daily_sales_report")
#> # A tibble: 8 x 3
#>   sales_quarter total_sales report_date
#>           <int>       <dbl> <date>
#> 1             1      28168. 2021-01-01
#> 2             2      30697. 2021-01-01
#> 3             4      27559. 2021-01-01
#> 4             3      26419. 2021-01-01
#> 5             3      26419. 2021-01-02
#> 6             2      30697. 2021-01-02
#> 7             1      28168. 2021-01-02
#> 8             4      27559. 2021-01-02
```

We can remove a partition and its underlying data using `delete_partitions_and_data` which uses an expression to match partitions - see https://boto3.amazonaws.com/v1/documentation/api/latest/reference/services/glue.html#Glue.Client.get_partitions for more details.

```
delete_partitions_and_data("new_db_dbtools", "daily_sales_report",
                           "report_date = '2021-01-02'")
read_sql_query("select * from new_db_dbtools.daily_sales_report")
#> # A tibble: 4 x 3
#>   sales_quarter total_sales report_date
#>           <int>       <dbl> <date>
#> 1              1       28168. 2021-01-01
#> 2              2       30697. 2021-01-01
#> 3              4       27559. 2021-01-01
#> 4              3       26419. 2021-01-01
```

Similarly we can remove a table and its data,

```
delete_table_and_data("new_db_dbtools", "daily_sales_report")
```

or the whole database.

```
delete_database_and_data("new_db_dbtools")
```