Vertica.dplyr User Guide

Edward Ma

2015-05-08

Vertica.dplyr

Vertica.dplyR is an R package developed by HP that provides Vertica-backend support for the dplyr package. Besides support of standard dplyr functionality (such as table manipulation), vertica.dplyr also features:

- HPdata-style functions (data transport from Vertica to Distributed R via the data loader), namely tbl2d**object** (where **object** is either **array** or **frame**), that are compatible with dplyr tbl objects.
- Easy CSV-loading into Vertica.
- copy_to functionality that takes advantage of Vertica's fast **COPY LOCAL** feature.
- Connectivity to Vertica through either JDBC **OR** ODBC.
- (Future Feature) Seamless Vertica UDx invocation.

If you are already familiar with what dplyr is and what it can do, you should skip the background description below.

Background

In the Vertica-R-DistributedR data-science package, one of the recurring user stories involves the ability of the R-proficient data scientist to have full and easy access to data and functionality that are in-database, and doing so from within the familiar environment of R.

Though several packages exist that allow the user to make a database connection and issue SQL statements from within the R console, many users are not comfortable with database syntax and lingo. In other cases, the user is indeed proficient in both R and SQL, but simply does not want to expend the effort to switch back and forth between these modes of instruction, because to do that is (as dplyr-author Hadley Wickham noted) "cognitively challenging".

Fortunately, there is a package in R that addresses this very common scenario. Developed by RStudio member and statistician Hadley Wickham, it is called dplyr.

What is dplyr?

Disclosure: This is just a brief introduction that will not do justice to the full extent of what dplyr can do. To learn more, please read the dplyr vignette.

The tool offers convenient methods of manipulating data frames, which can be analogized to tables in a database. Common actions in dplyr include what are known as the "verbs" of the package.

For single data.frames (or tables):

- **filter**
- Filters tables by rows matching certain criteria (e.g., filter(table,col1 > 5, col2 ==3))
- Equivalent to (SELECT ... from table WHERE ...)
- **arrange**

- Orders rows (e.g., arrange(tbl,desc(col1)))
- **select**
- Selects columns (e.g., select(tbl,'col1','col2'))
- **mutate**
- Create new columns (e.g., mutate(tbl,new_col1=col3/col2,new_col2=fun(col4,col5)))
- **summarise** (or **summarize**)
- Collapses rows belonging to a certain group into a single one (e.g., for aggregations)
- Analogous to GROUP BY

There are also verbs for multiple tables, such as (whose functions are mostly self-explanatory) left_join(), inner_join(), semi_join(), anti_join(), union(), intersect().

Vertica Integration

For data that are resident in a database, dplyr provides a convenient interface to access tables and manipulate them directly inside the database as though they were local data frames. Wickham has provided (natively, within the dplyr package itself) support for MySQL, Postgres, BigQuery, and SQLite backends. Vertica.dplyr enables dplyr to work with Vertica.

Since its inception, the package's scope has grown to be more than just small wrappers to get Vertica to work with standard dplyr functionality. Taken to maturity, Vertica.dplyr will not only add dplyr-support for Vertica, but also serve as the basic "intermediary" between R (and indeed, Distributed R) and Vertica, which will include data preparation, data transfer, and Vertica-function invocation (including UDxes in a future release).

Obtaining the Package

You can obtain vertica.dplyr from Github, and in the future, from CRAN.

Prerequisites

At present, vertica.dplyr has 2 hard dependencies: assertthat and dplyr. There are additional **soft** dependencies on vRODBC, RJDBC, and HPdata.

Soft means that there is no enforcement of having these packages installed on the user's system to install vertica.dplyr. However, to use vertica.dplyr, at least one of **RJDBC** or **vRODBC** must be installed and configured. These **soft** dependencies will be loaded (at which point it will be required to have the package installed) when their specific functionality is invoked. For example, when creating a vertica.dplyr connection to Vertica with ODBC, vertica.dplyr will check to make sure that **vRODBC** is installed and loadable. If it isn't, it will throw an error.

You will also need access to a Vertica database (must be configured separately), as well as either (or both) the ODBC or JDBC Vertica driver(s) installed and configured on your host machine (the one that will run vertica.dplyr). Please refer to external documentation for help with these steps.

Using Vertica.dplyr

Let's now walk through a simple example to illustrate some of the fundamental data manipulation features of vertica.dplyr (and dplyr).

First, load the package in R:

```
# Load the vertica.dplyr package, which loads dplyr as a dependency
library(vertica.dplyr)
```

Note that a few of the imported dplyr functions will have names that overwrite their R-base equivalents.

Connect to Vertica

First, you will need to connect to Vertica using the `src_vertica` function. There are two ways to do this:

Via ODBC (requires vRODBC):

```
# Connect to Vertica using vRODBC
vertica_odbc <- src_vertica("VerticaDSN")
```

Via JDBC (requires RJDBC):

```
# Connect to Vertica using RJDBC
vertica_jdbc <- src_vertica(dsn = NULL, jdbcpath="/opt/vertica/java/lib/vertica_jdbc.jar","foobar","loca
```

Here, I am connecting to a local Vertica DB instance named "foobar" that is listening on port 5433, with db-owner username "dbadmin", and no password. Notice that the ODBC version is much more terse, because all of the configuration details are settled inside of ODBC.ini on the system, and I only need to provide the DSN. Note that if `dsn` is NULL, it is assumed that JDBC is to be used.

Both `vertica_jdbc` and `vertica_odbc` are `src_vertica` objects, and vertica.dplyr functionality will work on both object types interchangeably. From the user-experience perspective, the only differences are how the package prints some metadata when you look at these objects,

```
# show differences
vertica_odbc
```

```
## src:  Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## tbls: AllstarFull, Appearances, AwardsManagers, AwardsPlayers,
##   AwardsShareManagers, AwardsSharePlayers, BattingPost,
##   compute_test_table, d1, d2, Fielding, FieldingOF, FieldingPost,
##   HallOfFame, LahmanData, Managers, ManagersHalf, Master, Pitching,
##   PitchingPost, ref_join, Salaries, Schools, SchoolsPlayers, SeriesPost,
##   Teams, TeamsFranchises, TeamsHalf, temp
```

```
vertica_jdbc
```

```
## src:  Vertica JDBC Connection [dbadmin@localhost:5433/foobar]
## tbls: AllstarFull, Appearances, AwardsManagers, AwardsPlayers,
##   AwardsShareManagers, AwardsSharePlayers, BattingPost,
```

```
##   compute_test_table, d1, d2, Fielding, FieldingOF, FieldingPost,
##   HallOfFame, LahmanData, Managers, ManagersHalf, Master, Pitching,
##   PitchingPost, ref_join, Salaries, Schools, SchoolsPlayers, SeriesPost,
##   Teams, TeamsFranchises, TeamsHalf, temp
```

as well as whether or not `dsn` is required as a parameter for `tbl2dframe` and `tbl2darray` (described in a later section).

As you can see, both of them print information about the connection and list the current user tables (and views) found in the database. For the rest of this tutorial, let's just use the ODBC version and call it `vertica`:

```
vertica <- vertica_odbc
class(vertica)
```

```
## [1] "src_vertica" "src_sql"     "src"
```

Vertica.dplyr "tables" (tbl_vertica)

Before we can walk through the example, we must understand what a "tbl" is. Some basic principles:

- The basic dplyr object is the "tbl" (or in this case, tbl_vertica). This is a reference to SQL, some not-yet executed query that will result in a new table.
- Conceptually, a "tbl_vertica" object is nothing more than a wrapper object that contains a SQL query, a connection to a DB, and related internal metadata. These tables behave in the same manner as data.frames would when used in dplyr, except now they are actually meant to be executed in database.
- Most operations (excluding those that convert tbls to data.frames, or saves them to tables/view in Vertica, or converts them to dobjects in Distributed R) performed on a tbl will result in another tbl. Essentially, you can chain these operations together, meaning that the SQL statement builds up in complexity.
- Tbl objects are lazily executed on the DB; that is, when you do not require the results (e.g., moving the results to R, or looking at the data by printing the results in R), nothing is executed in Vertica.

Retrieving a tbl reference

One way to get started with a tbl_vertica object is to directly access an existing table. For example, we could get access to the "Salaries" table listed above in the printout of our src_vertica objects. To do this, we use the `tbl` function on our `src_vertica` object.

```
salaries <- tbl(vertica,"Salaries")
```

`salaries` now contains a simple select statement on "Salaries" in Vertica. You can examine the SQL query associated with a `tbl_vertica` by accessing the `query` member of the object:

```
salaries$query
```

```
## <Query> SELECT "yearID", "teamID", "lgID", "playerID", "salary"
## FROM "Salaries"
## An object of class "VerticaConnection"
## Slot "conn":
```

```
## vRODBC Connection 6
## Details:
##   case=nochange
##   DSN=VerticaDSN
##
## Slot "type":
## [1] "ODBC"
```

Note that as mentioned before, nothing is yet executed in DB! If we would like to take a 'peek' at what the data look like, we could ask R to print `salaries`.

```
salaries
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: Salaries [23,956 x 5]
##
##    yearID teamID lgID  playerID salary
## 1    1985    ATL   NL barkele01 870000
## 2    1985    ATL   NL bedrost01 550000
## 3    1985    ATL   NL benedbr01 545000
## 4    1985    ATL   NL  campri01 633333
## 5    1985    ATL   NL ceronri01 625000
## 6    1985    ATL   NL chambch01 800000
## 7    1985    ATL   NL dedmoje01 150000
## 8    1985    ATL   NL forstte01 483333
## 9    1985    ATL   NL garbege01 772000
## 10   1985    ATL   NL harpete01 250000
## ..    ...    ...  ...       ...    ...
```

Now the query is actually executed in the database, and the first few rows can be seen in the result (dplyr automatically takes the HEAD and displays it).

Accessing data already in database is the presumed primary use-case. Another way to start our workflow (which is what we will use in the example in the following few sections) is to start with data in R and copy it to Vertica using `copy_to` (you can also start from a CSV file and load it into Vertica using `db_load_from_file`).

We start with data from all NYC flights in the year 2013, which we'll get from the R package nycflights13 (also maintained by Hadley Wickham). Importing the library imports an R data.frame, `flights`.

```
library(nycflights13)
# Peek at data.frame `flights`
head(flights,10)
```

```
## Source: local data frame [10 x 16]
##
##    year month day dep_time dep_delay arr_time arr_delay carrier tailnum
## 1  2013     1   1      517         2      830        11      UA  N14228
## 2  2013     1   1      533         4      850        20      UA  N24211
## 3  2013     1   1      542         2      923        33      AA  N619AA
## 4  2013     1   1      544        -1     1004       -18      B6  N804JB
```

```
## 5  2013     1   1      554        -6      812       -25      DL  N668DN
## 6  2013     1   1      554        -4      740        12      UA  N39463
## 7  2013     1   1      555        -5      913        19      B6  N516JB
## 8  2013     1   1      557        -3      709       -14      EV  N829AS
## 9  2013     1   1      557        -3      838        -8      B6  N593JB
## 10 2013     1   1      558        -2      753         8      AA  N3ALAA
## Variables not shown: flight (int), origin (chr), dest (chr), air_time
##   (dbl), distance (dbl), hour (dbl), minute (dbl)
```

```
nrow(flights)
```

```
## [1] 336776
```

Now let's copy it over to Vertica using `copy_to` and see what we get:

```
flights_vertica <- copy_to(vertica,flights,"flights")
flights_vertica
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: flights [336,776 x 16]
##
##    year month day dep_time dep_delay arr_time arr_delay carrier tailnum
## 1  2013     1   1      517         2      830        11      UA  N14228
## 2  2013     1   1      533         4      850        20      UA  N24211
## 3  2013     1   1      542         2      923        33      AA  N619AA
## 4  2013     1   1      544        -1     1004       -18      B6  N804JB
## 5  2013     1   1      554        -6      812       -25      DL  N668DN
## 6  2013     1   1      554        -4      740        12      UA  N39463
## 7  2013     1   1      555        -5      913        19      B6  N516JB
## 8  2013     1   1      557        -3      709       -14      EV  N829AS
## 9  2013     1   1      557        -3      838        -8      B6  N593JB
## 10 2013     1   1      558        -2      753         8      AA  N3ALAA
## ..  ...   ... ...      ...       ...      ...       ...     ...     ...
## Variables not shown: flight (dbl), origin (chr), dest (chr), air_time
##   (dbl), distance (dbl), hour (dbl), minute (dbl)
```

`copy_to` has automatically extracted the data types in the data.frame, converted them to a schema in Vertica, created the table in Vertica, and copied it over to the DB. It returns another tbl_vertica object, which we have named `flights_vertica`.

Small "data-prep" example using our flights data

As one of the most densely populated regions in the United States, the New York metropolitan area is serviced by three major airports: JFK, Newark (EWR), and LaGuardia (LGA). Delays in flights are unavoidable, but do any of these airports have an edge over the others? How is flight delay affected by the time of year?

To help me answer these inquiries, we may conduct a simple statistical analysis of the flight records in the year 2013.

Examining the `flights_vertica` table, we see that it has labeled each flight by (among other things) some values such as date, departure/arrival times, and distance.

Select: Select columns from a table.

The most straightforward and basic operation is the ***select*** function. When connected to a database, this will issue a corresponding SQL `SELECT` on the specified table, with the specified columns.

In our case, since we are only analyzing flight delay, the airports, and how these correspond to the dates (year is included but it's all 2013 so we'll discard the year column as well), we can simply select only those columns for now:

```
# Select columns month, day, arr_delay, and origin
q1 <- select(flights_vertica, month, day, origin, arr_delay)
q1
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: flights [336,776 x 4]
##
##    month day origin arr_delay
## 1      1   1    EWR        11
## 2      1   1    LGA        20
## 3      1   1    JFK        33
## 4      1   1    JFK       -18
## 5      1   1    LGA       -25
## 6      1   1    EWR        12
## 7      1   1    EWR        19
## 8      1   1    LGA       -14
## 9      1   1    JFK        -8
## 10     1   1    LGA         8
## ..   ... ...    ...       ...
```

Filter: Select rows matching provided criteria

To answer the first question (determining which airport has the worst delays), I want to examine only the flights that occur between the months of February and November. During the holiday season (which I'm approximating to the months of December and January), delays may be influenced by the increase in traveling, and such data may unfairly bias our result.

Since the `select` operation from before returned another dplyr tbl object, I can chain what I did there with the following `filter` operation by passing in `q1` as its first argument.

```
# Filter out flights in January and December
q2 <- filter(q1, year == 2013, month > 1, month < 12)
q2
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: flights [281,637 x 4]
```

```
## Filter: year == 2013, month > 1, month < 12
##
##    month day origin arr_delay
## 1      2   1    LGA        NA
## 2      2   1    EWR        NA
## 3      2   1    EWR        NA
## 4      2   1    EWR        NA
## 5      2   1    EWR        NA
## 6      2   1    EWR        NA
## 7      2   1    EWR        NA
## 8      2   1    EWR        NA
## 9      2   1    EWR        NA
## 10     2   1    EWR        NA
## ..   ... ...    ...       ...
```

The data now printed out show flights beginning with the first of February. Just as when we retrieved a reference to a Vertica table, we can view the query associated with this filter operation:

```
# Look at what we did in terms of SQL
q2$query
```

```
## <Query> SELECT "month" AS "month", "day" AS "day", "origin" AS "origin", "arr_delay" AS "arr_delay"
## FROM "flights"
## WHERE "year" = 2013.0 AND "month" > 1.0 AND "month" < 12.0
## An object of class "VerticaConnection"
## Slot "conn":
## vRODBC Connection 6
## Details:
##   case=nochange
##   DSN=VerticaDSN
##
## Slot "type":
## [1] "ODBC"
```

As you might have expected, the query resulting from a **filter** involved a `WHERE` clause.

You may notice that `arr_delay` has NA values for many rows, which are NULL in Vertica. We'll want to get rid of these rows with incomplete data using another `filter` operation on `q2`:

```
# Filter out NA rows
q3 <- filter(q2,!is.na(arr_delay))
q3
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: flights [273,928 x 4]
## Filter: year == 2013, month > 1, month < 12, !is.na(arr_delay)
##
##    month day origin arr_delay
## 1      2   1    EWR         4
```

```
## 2      2   1    EWR        -4
## 3      2   1    LGA         8
## 4      2   1    JFK       -10
## 5      2   1    JFK         9
## 6      2   1    EWR       -14
## 7      2   1    JFK        -1
## 8      2   1    LGA         9
## 9      2   1    LGA         0
## 10     2   1    LGA        -4
## ..   ... ...    ...       ...
```

We do not have NA values anymore. dplyr has converted our R-style boolean expression on `arr_delay` for `NA` values into `NOT NULL` in SQL.

Summarise, group_by, and arrange: GROUP BY airports and ORDER them by average delay

Now for the fun part. Our data have been cleaned up a bit, and we want to see which airport had the worst delays going out of NYC in 2013, between the months of February and December.

Astute SQL experts will recognize that these operations correspond to running AVG and GROUP BY operations on these columns. Of course, we'll want to use the dplyr equivalent.

The first thing to use is `group_by`, which constructs the grouping clause, followed by a summarise (which takes a group_by clause as its first argument) to generate our report. Along with our report, we would also like to create new columns for determining the number of flights belonging to each airport (we'll call this `count`), as well as the average delay for that airport (called delay).

```
# Group by airport (origin)
by_origin <- group_by(q3, origin)
q4 <- summarise(by_origin,
   count = n(),
   delay = mean(arr_delay)
)
```

The `summarise` function has allowed me to create two new columns, `count` and `delay` as functions of the `GROUP BY origin` created by `group_by`, which I stored as `by_origin`. The arguments on the right-hand side, `n` and `mean`, correspond to the vertica.dplyr bindings of the Vertica analytic functions I am invoking (**COUNT(*) **and** AVG(arr_delay)**).

Before I look at the result, let's finally rank these airports by worst (most delay) to best using `arrange`:

```
# Average delays by airport
q5 <- arrange(q4,desc(delay))
q5
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: <derived table> [?? x 3]
## Arrange: desc(delay)
##
##    origin count    delay
```

```
## 1     EWR 98101 7.733132
## 2     LGA 84702 5.370121
## 3     JFK 91125 5.268258
## ..    ...   ...      ...
```

Pretty interesting stuff. From these results we see that, compared to other airports servicing the NYC region, EWR (Newark) had, by roughly two-and-a-half minutes, the worst average arrival delay in 2013.

Lazy execution

Now let's look again at the query by accessing `$query` of a `tbl` object. We'll do it on our `q5` variable.

```
q5$query
```

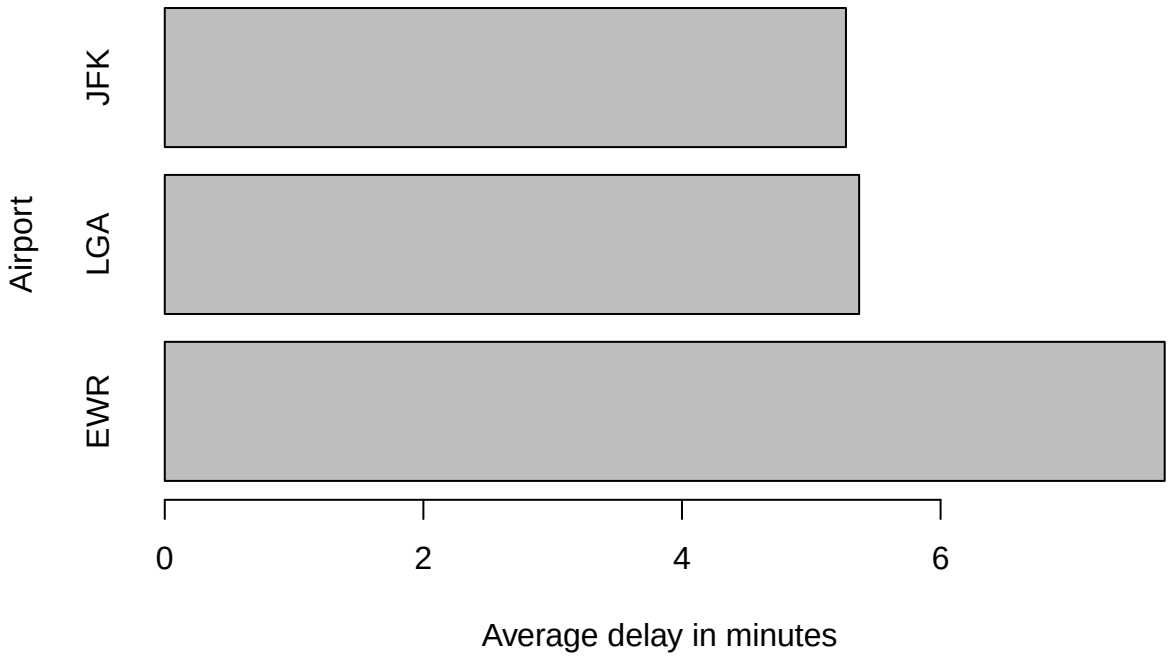
```
## <Query> SELECT "origin", "count", "delay"
## FROM (SELECT "origin", count(*) AS "count", AVG("arr_delay") AS "delay"
## FROM "flights"
## WHERE "year" = 2013.0 AND "month" > 1.0 AND "month" < 12.0 AND NOT("arr_delay"IS NULL)
## GROUP BY "origin") AS "_W10"
## ORDER BY "delay" DESC
## An object of class "VerticaConnection"
## Slot "conn":
## vRODBC Connection 6
## Details:
##   case=nochange
##   DSN=VerticaDSN
##
## Slot "type":
## [1] "ODBC"
```

Note that this query is a complex query that has been built off of our initial tbl created by our `copy_to` statement since before we even started our data prep. What this means is that if I hadn't been printing all of the intermediate results in console, *nothing from q1 through q5 would have even touched the DB.*

Collect: Bringing it back to R

What happens if I want to bring this back into the R console? Very easy, as it turns out, with `collect`. `collect` takes the saved query, executes it on the database, and brings the results back into R as the familiar data.frame.

```
# Bring results back into R as a data.frame, then plot the results
airport_delays <- collect(q5)
barplot(airport_delays[["delay"]],horiz=TRUE,
      names.arg=airport_delays[["origin"]],
     xlab="Average delay in minutes",ylab="Airport")
```



##

Calling Vertica Functions

In the previous sections, we saw the use of AVG (called as mean()) in dplyr) as well as COUNT(*) (invoked with n() in dplyr) with the `summarise` function. In this section, we'll cover in more formal detail how Vertica functions can be invoked, namely window and aggregate functions, on different partitions of data.

Note that a complete list of currently supported functions may be found at the end of this document.

In general, a function can be invoked by simply dropping it into any `mutate` or `summarise` statement. `mutate` adds an entire column to your tbl, so use it when you expect to get a vector of results back from your function, i.e., when invoking a window function, whereas `summarise` collapses all rows (belonging to a group) into one (therefore, if no `group_by` is used, your function will run on all rows of your table).

Aggregate Functions

An aggregate function needs to be invoked with `summarise`, which necessitates the complementary use of `group_by` to specify the groups on which to aggregate. If it is not used with any `group_by` result, the function will run on the entire table, leaving you with only one result.

More generally, if you have a function `foo` that you wish to run on attribute `attr` in `tbl`, grouping by column `col`, with additional arguments `args` you would run:

```
grouped_by_col <- group_by(tbl,col)
result <- summarise(grouped_by_col,new_col = foo(attr,args))
```

Occasionally, the function may not take the `attr` argument, or any arguments at all (such as COUNT(*)). In this case, no args need to be passed, e.g., `new_col = n()` for `COUNT(*)`.

Notice that a new column will be created named `new_col`. This can be done using `mutate` as well.

Window Functions

Currently, the **PARTITION BY** SQL statement is also constructed by `group_by`. This means that for window functions requiring such a clause (which is most), it is recommended to run a `group_by` on the column

you wish to partition by (although, as will be described later, this can be overridden using the `partition` parameter).

In addition to this, many window functions also have some notion of a *"window"* of a specified width over which the function is applied. Vertica is also special in that many of them (even those that do not require explicit ordering) require an obligatory ORDER BY clause when they are invoked.

For this reason, the syntax for running a window function on Vertica through dplyr is unique to vertica.dplyr.

In standard dplyr syntax, what is needed is a `group_by` call followed by a `mutate` or `filter`. If you wish to run a window function `winfun` on column `col` of table `tbl`, partitioning by `partcol`, you would invoke it in the following manner:

```
grouped_by_partcol <- group_by(tbl,partcol)
mutate(grouped_by_partcol, new_col = winfun(col,args))
```

You may notice that this does not enforce any particular ordering nor windowing range on the function. Because most Vertica functions require the **ORDER BY** statement, in vertica.dplyr there is a convenient way to make this happen in your invocation of the window function, using the `order` argument.

Similarly, for convenience, you can specify your window size using `range`. This is usually an optional argument (leaving this alone will attempt to run the function with no window size specified), but it is useful when it is needed.

Using both of these arguments, with the assumption that we want to have the range between `min` and `max`, ordering by `ordercol`, the above flow is converted to the following in vertica.dplyr:

```
grouped_by_partcol <- group_by(tbl,partcol)
mutate(grouped_by_partcol, new_col = winfun(col,args,range=c(min,max),order=ordercol))
```

The following will also have the same effect, since `partition` will also allow you to specify partitioning:

```
mutate(tbl, new_col = winfun(col,args,range=c(min,max),partition=partcol,order=ordercol))
```

Note that the range is specified as an R vector. `-Inf` is converted to `ROWS UNBOUNDED PRECEDING`, `Inf` is converted to `ROWS UNBOUNDED FOLLOWING`.

Hands-on Example Let us return to our flights example, and grab our query `q1`. This time, we want to find out how delays changed over the calendar year. Let's conceive of a flow to achieve this:

1. Remove unnecessary columns and NULL values
2. Average all delays for every unique calendar day
3. To get a more "smoothed" result, run another average function over these delay values, with a running window width of 10 days (5 days prior to the current row, and 5 days after).

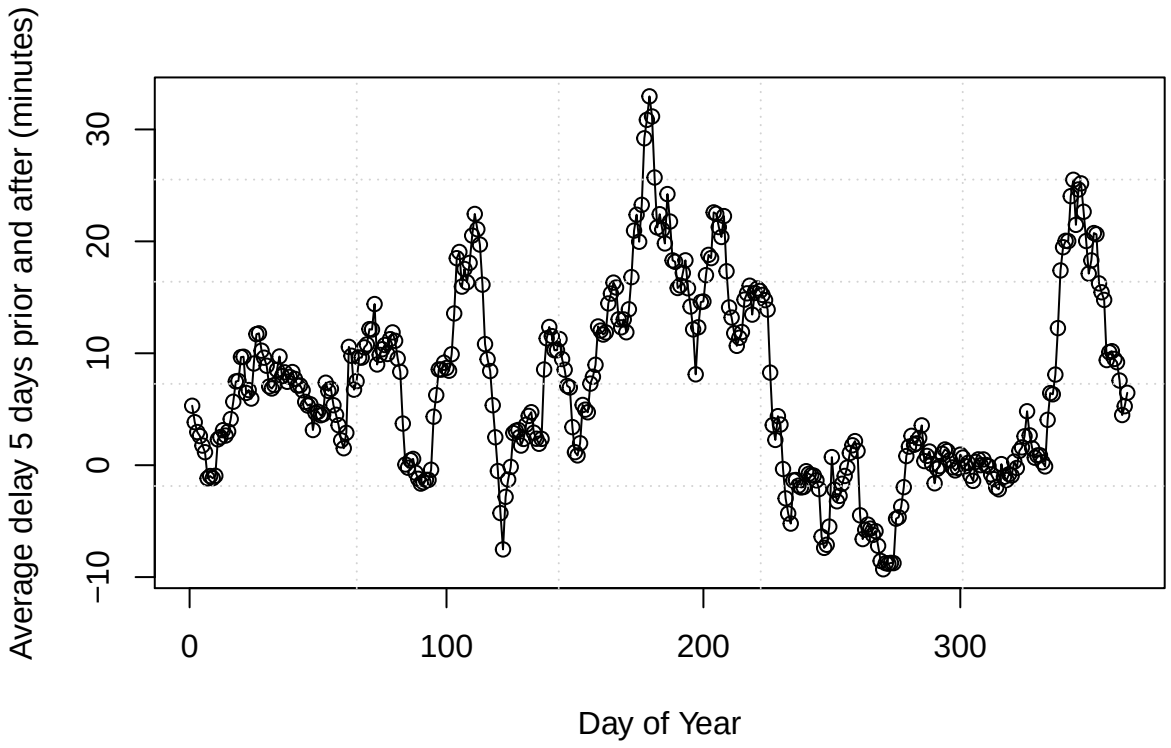
```
c <- filter(select(q1,month,day,arr_delay),!is.na(arr_delay))
calendar_day <- group_by(c,month,day)
delay_by_day <- summarise(calendar_day,avg_delay = mean(arr_delay))
delays <- mutate(delay_by_day, delay_smoothed = mean(avg_delay,range=c(-5,5),
partition=NULL,order=c(month,day)))
delays
```

```
## Source: Vertica ODBC Connection
## -----+DSN: VerticaDSN
## -----+Host: 127.0.0.1
## -----+DB Version: 07.01.0001
## -----+ODBC Version: 03.52
## From: <derived table> [?? x 4]
## Grouped by: month
##
##    month day  avg_delay delay_smoothed
## 1      1   1 12.6510229      5.3091754
## 2      1   2 12.6928879      3.8439629
## 3      1   3  5.7333333      2.9600202
## 4      1   4 -1.9328194      2.6017649
## 5      1   5 -1.5258020      1.7517068
## 6      1   6  4.2364294      1.1595273
## 7      1   7 -4.9473118     -1.1738523
## 8      1   8 -3.2275785     -0.9703107
## 9      1   9 -0.2642777     -1.1605861
## 10     1  10 -5.8988159     -0.9461796
## ..   ... ...        ...            ...
```

Notice that we needed to set `partition=NULL` to clear the implicit `PARTITION BY MONTH,DAY` that would've been there from our earlier `group_by`.

Now let's plot `delay_smoothed`.

```
delays_local <- collect(delays)
plot(delays_local[["delay_smoothed"]],xlab="Day of Year",
ylab="Average delay 5 days prior and after (minutes)")
lines(delays_local[["delay_smoothed"]])
grid(nx=5)
```



Delays in flights appear to peak during the summer months, with smaller peaks during the winter months (holiday season), which somewhat follows intuition. Interestingly, between days 200 and 300 (August to November), flights are very on time, and even have a tendency to be a little early!

For a list of functions that are currently tested to work in vertica.dplyr and their mappings to internal Vertica functions, please refer to the appropriately titled section at the end of this document.

Note: Beware of types. By default, numbers passed into R functions are of type numeric, and these will not work with Vertica functions that require integer values. To pass integer values, you must use "L" after the number in R, i.e., `func(2L,3L)`.

Other (handy) functionality

We've gone over accessing tables and manipulating them with most of the operations that likely describe the majority of what you will do with vertica.dplyr, including a general template for how to invoke Vertica functions.

Now we'll (in a much more brief way) go over some of the additional things that can be done using vertica.dplyr.

Many of the functions described below will take a `VerticaConnection` object, meaning something that may be accessed by retrieving the `con` member of your `src_vertica` object. In the examples that we've been using, this would be `vertica$con`.

Creating a table in Vertica

More often than not, the data to be prepared already exist in Vertica. On some other occasions, you may want to copy data from your R instance to Vertica using `copy_to`. Other times, you might want to create a table of your own and take control over your schema and data types.

To do so, you need to create a character vector that describes the schema of your table, and invoke `db_create_table` with a `VerticaConnection`.

For example, to create a table named `New_Table`, with a schema of `INTEGER`, `VARCHAR`, and `VARCHAR` corresponding to columns `employeeID`, `name`, and `department`, I would be able to do so in the following way:

```
types <- c("integer","varchar","varchar")
names(types) <- c("employeeID","name","department")
db_create_table(vertica$con,"New_Table",types)
```

Loading from CSV

`db_load_from_file` provides a convenient way to load data from a file. To invoke it, you would need to know:

1. The name of the file (*example: foo.csv*)
2. Your `src_vertica` object (*example: `vertica`*)
3. The type of delimiter in the file (*example: ','*)
4. The number of lines to skip at the beginning of the file, which is useful for files with headings (*example: 1L*)
5. Whether you would like to append this data to existing data in a table or overwrite it. (*example: TRUE*)
6. The name of the table into which you would like to load the data (*example: mytable*)

```
db_load_from_file(vertica,"mytable","foo.csv",sep=",",skip=1L,append=TRUE)
```

Note that `mytable` will already have to exist, and the schema will have to be compatible with the data you are loading. At the moment, vertica.dplyr will **not** automatically scan your file for a schema and create an appropriate table for you.

Listing, dropping, and checking tables

You can view, check for, and drop tables by using `db_list_tables`, `db_has_table`, and `db_drop_table`, respectively:

```
db_list_tables(vertica$con)
```

```
##  [1] AllstarFull         Appearances         AwardsManagers
##  [4] AwardsPlayers       AwardsShareManagers AwardsSharePlayers
##  [7] BattingPost         compute_test_table  d1
## [10] d2                  Fielding            FieldingOF
## [13] FieldingPost        flights             HallOfFame
## [16] LahmanData          Managers            ManagersHalf
## [19] Master              Pitching            PitchingPost
## [22] ref_join            Salaries            Schools
## [25] SchoolsPlayers      SeriesPost          Teams
## [28] TeamsFranchises     TeamsHalf           temp
## 30 Levels: AllstarFull Appearances AwardsManagers ... temp
```

```
db_has_table(vertica$con,"foobar")
```

```
## [1] FALSE
```

```
db_has_table(vertica$con,"Salaries")
```

```
## [1] TRUE
```

```
db_drop_table(vertica$con,"Salaries")
```

Explaining Queries

Queries (tbl objects) can be explained using `explain`:

```
explain(q5)
```

```
## <SQL>
## SELECT "origin", "count", "delay"
## FROM (SELECT "origin", count(*) AS "count", AVG("arr_delay") AS "delay"
## FROM "flights"
## WHERE "year" = 2013.0 AND "month" > 1.0 AND "month" < 12.0 AND NOT("arr_delay"IS NULL)
## GROUP BY "origin") AS "_W10"
## ORDER BY "delay" DESC
##
```

```
##
## <PLAN>
## ------------------------------ QUERY PLAN DESCRIPTION: ------------------------------
## EXPLAIN SELECT "origin", "count", "delay" FROM (SELECT "origin", count(*) AS "count", AVG("arr_delay
## Access Path:+-SORT [Cost: 220K, Rows: 3] (PATH ID: 1)
## |  Order: _W10.delay DESC
## | +---> GROUPBY HASH (LOCAL RESEGMENT GROUPS) [Cost: 220K, Rows: 3] (PATH ID: 3)
## | |      Aggregates: max(flights.origin), count(*), sum_float(flights.arr_delay), count(flights.arr_
## | |      Group By: collation(flights.origin, 'en_US')
## | | +---> STORAGE ACCESS for flights [Cost: 219K, Rows: 280K] (PATH ID: 4)
## | | |      Projection: public.flights_super
## | | |      Materialize: flights.arr_delay, flights.origin
## | | |      Filter: (flights.year = 2013.0)
## | | |      Filter: ((flights.month > 1.0) AND (flights.month < 12.0))
## | | |      Filter: (NOT (flights.arr_delay IS NULL))
```

Saving queries to views and tables

`compute` and `db_save_view` will allow you to save your `tbl` objects to tables and views in Vertica, respectively. Once saved as a view or table, they can easily be transported to Distributed R via `tbl2dframe` and `tbl2darray` (described in the following section).

```
# Save to a table named "mytable"
compute(tbl,name="mytable")
# Save to a view named "myview"
db_save_view(tbl,name="myview")
```

Using Distributed R's Native-Data-Loader with tbl2darray and tbl2dframe (requires vRODBC)

The Distributed-R team has developed a method of fast parallel transfer between the Vertica database and Distributed R. This technique utilizes several parallel ODBC connections, through which data are pushed from Vertica in a data-locality-aware manner (a **PARTITION BEST** UDx).

The functionality is available in the HPdata package, along with a standard ODBC connector interface. `HPdata` is another *soft* dependency of vertica.dplyr, meaning that vertica.dplyr will only attempt to load it when either `tbl2darray` or `tbl2dframe` is invoked.

Converting a `tbl_vertica` object to a Distributed R `dframe` or `darray` is easy. One only needs to provide a `tbl_vertica` object to either `tbl2darray` or `tbl2dframe` if using a vRODBC-backed tbl_vertica, or the `tbl_vertica` object and a `dsn` reference for ODBC if using a RJDBC-backed tbl_vertica.

You can transform a `tbl_vertica` that uses *either* vRODBC or RJDBC, but the Vertica connector uses vRODBC; ergo, an extra `dsn` argument is needed for the RJDBC case.

Additional parameters to `tbl2dframe` and `tbl2darray` exist, such as the number of partitions for the constructed dframe or darray; please see the manual page for details.

```
# Example 1, using vRODBC-backed tbl_vertica
foo <- tbl(vertica,"myTable")
my.dframe <- tbl2dframe(foo)

# Example 2, using a RJDBC-backed tbl_vertica
foo <- tbl(vertica,"myTable")
```

```
bar <- mutate(foo,col=col1+col2)
my.darray <- tbl2darray(bar,dsn="VerticaDSN",npartitions=4)
```

List of tested Vertica analytic functions and their Vertica.dplyr names

Functions that are not listed here have not been tested, but may be invokable by using the Vertica function name from within vertica.dplyr.

Vertica Function Name : Name in vertica.dplyr

- **lag** : lag
- **conditional_change_event** : changed
- **conditional_true_event** : isTrue
- **ntile** : ntile
- **lead** : lead
- **median** : median
- **row_number** : row_number
- **first_value** : head
- **last_value** : tail
- **exponential_moving_average** : exp_mvg_avg
- **percentile_cont** : percentile_cont
- **percentile_disc** : percentile_disc
- **rank** : rank, min_rank
- **dense_rank** : dense_rank
- **percent_rank** : percent_rank
- **cume_dist** : cume_dist
- **row_number** : row_num
- **stddev** : sd
- **stddev_pop** : sd_pop
- **var_pop** : var_pop
- **var_samp** : var_samp
- **avg** : mean
- **sum** : sum
- **min** : min
- **max** : max
- **count(*)** : n
- **regr_slope** : lm_slope
- **regr_intercept** : lm_intercept
- **covar_samp** : cov
- **bit_and** : bitwAnd
- **bit_or** : bitwOr
- **bit_xor** : bitwXor

For descriptions of and help for these functions, please refer to the appropriate section(s) in the Vertica User Manual.

Further Reading

This user guide touches upon most of the useful features in vertica.dplyr, but is not comprehensive.

For more information and help on the functions that vertica.dplyr exports, please refer to their manual pages.

For more literature on dplyr, you may look at the excellent documentation available for it in Hadley Wickham's RStudio pages.