I was chatting with a fellow Amazon Athena user and the topic of using Presto functions such as approx_distinct() via {d[b]plyr} came up and it seems it might not be fully common knowledge that any non-already translated function is passed to the destination intact. That means you can just "use" `approx_distinct()` and it will work just fine. Here's an example using the ODBC {DBI} interface:

```
library(dbplyr)
library(tidyverse)

# My personal Athena workgroup has been upgraded to "engine 2"
# so Presto 0.217 functions are available. Only noting that for
# folks who may not keep up with AWS announcements.
#
# https://prestodb.io/docs/0.217/index.html

DBI::dbConnect(
  odbc::odbc(),
  driver = "/Library/simba/athenaodbc/lib/libathenaodbc_sbu.dylib",
  Schema = "sampledb",
  AwsRegion = "us-east-1",
  AuthenticationType = "IAM Profile",
  AWSProfile = "personal",
  MaxCatalogNameLen = 0L,
  MaxSchemaNameLen = 0L,
  MaxColumnNameLen = 0L,
  MaxTableNameLen = 0L,
  UseResultsetStreaming = 1L,
  StringColumnLength = 32 * 1024L,
  S3OutputLocation = "s3://accessible-bucket/"
) -> con

# this comes with Athena
elb_logs <- tbl(con, "elb_logs")

elb_logs
## # Source:   table [?? x 16]
## # Database: Amazon Athena 01.00.0000[@Amazon Athena/AwsDataCatalog]
##    timestamp elbname requestip requestport backendip backendport
##
##  1 2014-09-… lb-demo 251.51.8…       17141 251.111.…        8000
##  2 2014-09-… lb-demo 244.201.…       17141 244.140.…        8888
##  3 2014-09-… lb-demo 242.204.…       17141 255.196.…        8888
##  4 2014-09-… lb-demo 251.51.8…       17141 255.129.…        8888
##  5 2014-09-… lb-demo 242.241.…       17141 255.129.…        8899
##  6 2014-09-… lb-demo 243.198.…       17141 255.129.…        8888
##  7 2014-09-… lb-demo 244.119.…       17141 242.89.1…          80
##  8 2014-09-… lb-demo 254.173.…       17141 251.51.8…        8000
##  9 2014-09-… lb-demo 243.198.…       17141 254.149.…        8888
## 10 2014-09-… lb-demo 249.185.…       17141 241.36.2…        8888
## # … with more rows, and 10 more variables: requestprocessingtime ,
```

```
## #   backendprocessingtime , clientresponsetime ,
## #   elbresponsecode , backendresponsecode ,
## #   receivedbytes , sentbytes , requestverb ,
## #   url , protocol

elb_logs %>%
  summarise(d = n_distinct(backendip)) # 0.62 seconds
## # Source:   lazy query [?? x 1]
## # Database: Amazon Athena 01.00.0000[@Amazon Athena/AwsDataCatalog]
##         d
##
## 1    2311

# https://prestodb.io/docs/0.217/functions/aggregate.html#approx_distinct

elb_logs %>%
  summarise(d = approx_distinct(backendip)) # 0.49 seconds
## # Source:   lazy query [?? x 1]
## # Database: Amazon Athena 01.00.0000[@Amazon Athena/AwsDataCatalog]
##         d
##
## 1    2386
```

In this toy example there's no real reason to use this alternate function, but on my datasets using the approximator version dramatically reduces query time, reduces query cost, and produces results that by default have a standard error of 2.3% (which is fine for the use-cases I apply this to). There's an alternate signature which lets you supply the standard error, as well.

If you're curious as to what functions are translated by default, just use `sql_translate_env()` on the connection object:

```
sql_translate_env(con)
##
## scalar:    -, :, !, !=, (, [, [[, {, *, /, &, &&, %/%, %%, %>%,
## scalar:    %in%, ^, +, <, <=, ==, >, >=, |, ||, $, abs, acos,
## scalar:    as_date, as_datetime, as.character, as.Date,
## scalar:    as.double, as.integer, as.integer64, as.logical,
## scalar:    as.numeric, as.POSIXct, asin, atan, atan2, between,
## scalar:    bitwAnd, bitwNot, bitwOr, bitwShiftL, bitwShiftR,
## scalar:    bitwXor, c, case_when, ceil, ceiling, coalesce, cos,
## scalar:    cosh, cot, coth, day, desc, exp, floor, hour, if,
## scalar:    if_else, ifelse, is.na, is.null, log, log10, mday,
## scalar:    minute, month, na_if, nchar, now, paste, paste0, pmax,
## scalar:    pmin, qday, round, second, sign, sin, sinh, sql, sqrt,
## scalar:    str_c, str_conv, str_count, str_detect, str_dup,
## scalar:    str_extract, str_extract_all, str_flatten, str_glue,
## scalar:    str_glue_data, str_interp, str_length, str_locate,
## scalar:    str_locate_all, str_match, str_match_all, str_order,
## scalar:    str_pad, str_remove, str_remove_all, str_replace,
## scalar:    str_replace_all, str_replace_na, str_sort, str_split,
## scalar:    str_split_fixed, str_squish, str_sub, str_subset,
## scalar:    str_to_lower, str_to_title, str_to_upper, str_trim,
```

```
## scalar:    str_trunc, str_view, str_view_all, str_which,
## scalar:    str_wrap, substr, substring, switch, tan, tanh, today,
## scalar:    tolower, toupper, trimws, wday, xor, yday, year
## aggregate: cume_dist, cummax, cummean, cummin, cumsum,
## aggregate: dense_rank, first, lag, last, lead, max, mean, median,
## aggregate: min, min_rank, n, n_distinct, nth, ntile, order_by,
## aggregate: percent_rank, quantile, rank, row_number, sd, sum, var
## window:    cume_dist, cummax, cummean, cummin, cumsum,
## window:    dense_rank, first, lag, last, lead, max, mean, median,
## window:    min, min_rank, n, n_distinct, nth, ntile, order_by,
## window:    percent_rank, quantile, rank, row_number, sd, sum, var
```

The release of the latest versions of {d[b]plyr} destroyed a lazy, bad, hack I was using to cast columns to JSON (you'll note the lack of a `cast()` function above, which is necessary for Athena since the syntax is not that of a function call). I'm _very_glad they did since it's bad to rely on undocumented functionality and, honestly, it's pretty straightforward to make an "official" translation for them.

First, we need the class of this Athena ODBC connection:

```
class(con)
## [1] "Amazon Athena"
## attr(,"package")
## [1] ".GlobalEnv"
```

We'll need to write a `sql_translation.Amazon Athena()` function for this connection class and we'll start with writing one that doesn't handle our casting just to show the basic setup:

```
`sql_translation.Amazon Athena` <- function(x) {
  sql_variant(
    dbplyr::base_odbc_scalar,
    dbplyr::base_odbc_agg,
    dbplyr::base_odbc_win
  )
}
```

All that function is doing (now) is setting up the default translators you've seen in the above output listings.

To make it do something else, we need to add casting translator helpers, which fall under the "scalar" category. This, too, is pretty straightforward since {dbplyr} makes it possible to just extend a parent set of category translators:

```
sql_translator(
  .parent = dbplyr::base_odbc_scalar,
  cast_as = function(x, y) dbplyr::build_sql("CAST(", x, " AS ", y,
")"),
  try_cast_as = function(x, y) dbplyr::build_sql("TRY_CAST(", x, " AS
", y, ")")
) -> athena_scalar

`sql_translation.Amazon Athena` <- function(x) {
  sql_variant(
    athena_scalar,
```

```
    dbplyr::base_odbc_agg,
    dbplyr::base_odbc_win
  )
}
```

Now, let's see if it *really* knows about our new casting functions:

```
sql_translate_env(con)
##
## scalar:    -, :, !, !=, (, [, [[, {, *, /, &, &&, %/%, %%, %>%,
## scalar:    %in%, ^, +, <, <=, ==, >, >=, |, ||, $, abs, acos,
## scalar:    as_date, as_datetime, as.character, as.Date,
## scalar:    as.double, as.integer, as.integer64, as.logical,
## scalar:    as.numeric, as.POSIXct, asin, atan, atan2, between,
## scalar:    bitwAnd, bitwNot, bitwOr, bitwShiftL, bitwShiftR,
## scalar:    bitwXor, c, case_when, cast_as, ceil, ceiling,
## scalar:    coalesce, cos, cosh, cot, coth, day, desc, exp, floor,
## scalar:    hour, if, if_else, ifelse, is.na, is.null, log, log10,
## scalar:    mday, minute, month, na_if, nchar, now, paste, paste0,
## scalar:    pmax, pmin, qday, round, second, sign, sin, sinh, sql,
## scalar:    sqrt, str_c, str_conv, str_count, str_detect, str_dup,
## scalar:    str_extract, str_extract_all, str_flatten, str_glue,
## scalar:    str_glue_data, str_interp, str_length, str_locate,
## scalar:    str_locate_all, str_match, str_match_all, str_order,
## scalar:    str_pad, str_remove, str_remove_all, str_replace,
## scalar:    str_replace_all, str_replace_na, str_sort, str_split,
## scalar:    str_split_fixed, str_squish, str_sub, str_subset,
## scalar:    str_to_lower, str_to_title, str_to_upper, str_trim,
## scalar:    str_trunc, str_view, str_view_all, str_which,
## scalar:    str_wrap, substr, substring, switch, tan, tanh, today,
## scalar:    tolower, toupper, trimws, try_cast_as, wday, xor,
## scalar:    yday, year
## aggregate: cume_dist, cummax, cummean, cummin, cumsum,
## aggregate: dense_rank, first, lag, last, lead, max, mean, median,
## aggregate: min, min_rank, n, n_distinct, nth, ntile, order_by,
## aggregate: percent_rank, quantile, rank, row_number, sd, sum, var
## window:    cume_dist, cummax, cummean, cummin, cumsum,
## window:    dense_rank, first, lag, last, lead, max, mean, median,
## window:    min, min_rank, n, n_distinct, nth, ntile, order_by,
## window:    percent_rank, quantile, rank, row_number, sd, sum, var
```

Aye! Let's test it out.

Unfortunately, this boring, default database has no `MAP` columns to really show this off, but we can convert a simple character column into JSON just to get the idea:

```
elb_logs %>%
  select(backendip)
## # Source:   lazy query [?? x 1]
## # Database: Amazon Athena 01.00.0000[@Amazon Athena/AwsDataCatalog]
##    backendip
##
##  1 249.6.80.219
```

```
##  2 248.178.189.65
##  3 254.70.228.23
##  4 248.178.189.65
##  5 252.0.81.65
##  6 248.178.189.65
##  7 245.241.133.121
##  8 244.202.183.67
##  9 255.226.190.127
## 10 246.22.152.210
## # … with more rows
```

```
elb_logs %>%
  select(backendip) %>%
  mutate(
    backendip = cast_as(backendip, JSON)
  )
## # Source:   lazy query [?? x 1]
## # Database: Amazon Athena 01.00.0000[@Amazon Athena/AwsDataCatalog]
##    backendip
##
##  1 "\"244.238.214.120\""
##  2 "\"248.99.214.228\""
##  3 "\"243.3.190.175\""
##  4 "\"246.235.181.255\""
##  5 "\"241.112.203.216\""
##  6 "\"240.147.242.82\""
##  7 "\"248.99.214.228\""
##  8 "\"248.99.214.228\""
##  9 "\"253.161.243.121\""
## 10 "\"248.99.214.228\""
## # … with more rows
```

FIN

Despite the {tidyverse} documentation being written with care and clarity, this part of the R ecosystem is so extensive and evolving that watching out for all the doors and corners can be tricky. It's easy for the short paragraph on the "untranslated function" capability to be overlooked and it may be hard to fully grok the translation concept without an IRL example.

Hopefully this helped (even if only a little) demystify these two areas of {d[b]plyr}.
