Trick question: is a 10,000 cell numeric data.frame big or small?

In the era of "big data" 10,000 cells is minuscule. Such data could be fit on fewer than 1,000 punched cards (or less than half a box).

The joking answer is: it is small when they are selling you the system, but can be considered unfairly large later.

**Example**

Let’s look at a few examples in R. First let’s set up our examples. A 10,000 row by one column data.frame (probably fairly close the common mental model of a 100,000 cell data.frame), and a 100,000 column by one row data.frame (frankly bit of an abuse, but data warehouse tables with millions of rows and 500 to 1,000 columns are not uncommon).

dTall <- as.data.frame(matrix(data = 0.0,

nrow = 10000,

ncol = 1))

dWide <- as.data.frame(matrix(data = 0.0,

nrow = 1,

ncol = 10000))

For our example problem we will try to select (zero) rows based on a condition written against the first column.

**Base R**

For standard R working with either data.frame is not a problem.

system.time(nrow(dTall[dTall$V1>0, , drop = FALSE]))

## user system elapsed

## 0 0 0

system.time(nrow(dWide[dWide$V1>0, , drop = FALSE]))

## user system elapsed

## 0.060 0.004 0.064

**dplyr**

For dplyr the tall frame is no problem, but the wide frame can take almost 5 minutes to filter.

library("dplyr")

##

## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':

##

## filter, lag

## The following objects are masked from 'package:base':

##

## intersect, setdiff, setequal, union

system.time(dTall %>% filter(V1>0) %>% tally())

## user system elapsed

## 0.059 0.003 0.061

system.time(dWide %>% filter(V1>0) %>% tally())

## user system elapsed

## 2.224 0.087 2.320

We will dig deeper into the dplyr timing on the wide table later.

**Databases**

Most databases don’t really like to work with a ridiculous number of columns.

**RSQLite**

RSQLite refuses to worm with the wide frame.

db <- DBI::dbConnect(RSQLite::SQLite(),

":memory:")

DBI::dbWriteTable(db, "dTall", dTall,

overwrite = TRUE,

temporary = TRUE)

DBI::dbWriteTable(db, "dWide", dWide,

overwrite = TRUE,

temporary = TRUE)

## Error in rsqlite\_send\_query(conn@ptr, statement): too many columns on dWide

DBI::dbDisconnect(db)

**RPostgres**

RPostgres refuses the wide frame, stating a hard limit of 1600 columns.

db <- DBI::dbConnect(RPostgres::Postgres(),

host = 'localhost',

port = 5432,

user = 'postgres',

password = 'pg')

DBI::dbWriteTable(db, "dTall", dTall,

overwrite = TRUE,

temporary = TRUE)

DBI::dbWriteTable(db, "dWide", dWide,

overwrite = TRUE,

temporary = TRUE)

## Error in result\_create(conn@ptr, statement): Failed to fetch row: ERROR: tables can have at most 1600 columns

DBI::dbDisconnect(db)

**sparklyr**

sparklyr fails, losing the cluster connection when attempting to write the wide frame.

spark <- sparklyr::spark\_connect(version='2.2.0',

master = "local")

DBI::dbWriteTable(spark, "dTall", dTall,

temporary = TRUE)

DBI::dbWriteTable(db, "dWide", dWide,

temporary = TRUE)

## Error in connection\_quote\_identifier(conn@ptr, x): Invalid connection

sparklyr::spark\_disconnect(spark)

**Why I care**

Some clients have run into intermittent issues on Spark at around 700 columns. One step of working around the issue was trying a range of sizes to try and figure out where the issue was and get a repeatable failure ( always an important step in debugging).

**Extra: dplyr again at larger scale.**

Let’s look a bit more closely at that dplyr run-time. We will try to get the nature of the column dependency by pushing the column count ever further up: to 100,000.

This is still less than a megabyte of data. It can fit on a 1986 era 1.44 MB floppy disk.

dWide <- as.data.frame(matrix(data = 0.0,

nrow = 1,

ncol = 100000))

dwt <- system.time(dWide %>% filter(V1>0) %>% tally())

print(dwt)

## user system elapsed

## 251.441 28.067 283.060

**Python**

For comparison we can measure how long it would take to write the results out to disk, start up a Python interpreter, use Pandas do do the work, and then read the result back in to R.

start\_pandas <- Sys.time()

feather::write\_feather(dWide, "df.feather")

import pandas

import feather

df = feather.read\_dataframe('df.feather')

print(type(df))

##

print(df.shape)

## (1, 100000)

df\_filtered = df.query('V1>1')

feather.write\_dataframe(df\_filtered, 'dr.feather')

res <- feather::read\_feather('dr.feather')

nrow(res)

## [1] 0

end\_pandas <- Sys.time()

python\_duration <- difftime(end\_pandas, start\_pandas,

unit = "secs")

print(python\_duration)

## Time difference of 21.47297 secs

ratio <- as.numeric(dwt['elapsed'])/as.numeric(python\_duration)

print(ratio)

## [1] 13.18216

This is slow, but still 13.2 times faster than using dplyr.