An intro to cloud native approaches for big data analysis

2022-11-08

Monitoring technology - from satellites to smart phones - is creating vast amounts of data every day. These data hold promise to provide global insights on everything from biodiversity patterns to vegetation change at increasingly fine spatial and temporal resolution. But leveraging this information often requires us to work with data that is too big to fit in our computer's "working memory" (RAM) or even to download to our computer's hard drive. Some data - like high resolution remote sensing imagery - might be too large to download even when we have access to big locally managed compute infrastructure.

In this post, I will walk through some tools, terms, and examples to get started with cloud native workflows. These workflows allow us to remotely access and query large data from online resources or web services, all while skipping the download step. We'll touch on a couple common data storage and cataloging structures (Parquet files, S3, and STAC) and tools to query these online data (Apache Arrow and Vsicurl). I'll start with tabular data example and then move on to spatial data - sharing resources along the way for deeper dives on each of the topics and tools.

Example code will be provided in R and case studies focused on environmental data, but these concepts are not specific to a programming language or research domain. I hope that this post provides a starting point for anyone interested in working with big data using cloud native approaches!

Tabular data: Querying files from cloud storage with Apache Arrow

Example: Plotting biodiversity observations through time

The Global Biodiversity Information Facilty (GBIF) synthesizes biodiversity data derived from a variety of sources - including everything from museum specimens to smartphone photos recorded on participatory science platforms like eBird and iNaturalist. The database includes over a billion species occurance records - probably not too big to download onto your machine's hard drive, but likely too big to read into your working memory. Downloading the whole database and then figuring out how to work with it seems like a drag, so we will walk through a cloud native approach to collect and download only the information we need from this database

Step 1. Finding the data online:

There are snapshots (i.e., versions) of the entire GBIF database on Amazon Web Services (AWS) S3 (you check them out and pick a version here). Lots of data is stored on AWS or other S3 compatible platforms (not just observation records of plants and animals), so the following workflow is not specific to GBIF.

What is S3? Simple Storage Service (S3) is just a structure of object storage through a web service interface. S3 compatibility is a requirement for cloud-native workflows (what we are working towards here) - cloud-native applications use the S3 API to communicate with object storage. The GBIF snapshot we leverage here is stored on Amazon Web Services (AWS) S3, but there are other s3 compatible object storage platforms (e.g., MinIO) that you can query with the same methods that we will explore below.

First, we will grab the S3 Uniform Resource Identifier (URI) for a recent GBIF snapshot (note the `s3://` placed before the URI name)

```
gbif_snapshot <- "s3://gbif-open-data-af-south-1/occurrence/2022-11-01/occurrence.parquet"
```

You might also notice that this is a `.parquet` file. *What is a Parquet file?* Parquet is an open source, column-oriented data file format designed for efficient data storage and retrieval. You can read all about it here. You can just think of it as a more efficient way to store tabular data - what you might otherwise store in a `.csv`.

We'll use the R package `arrow` which supports reading data sets from cloud storage without having to download them (allowing for large queries to run locally by only downloading the parts of the data-set necessary). You can read more about Apache Arrow here if you are interested in digging deeper into other applications (or interested in interfacing with the `arrow` package in other programming languages like python)! Note that arrow doesn't require that data is a parquet file - it also works with "csv", "text" and "feather" file formats.

```
library(arrow)
```

```
##
## Attaching package: 'arrow'
```

```
## The following object is masked from 'package:utils':
##
##     timestamp
```

```
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --
```

```
## v ggplot2 3.3.6      v purrr   0.3.5
## v tibble  3.1.8      v dplyr   1.0.10
## v tidyr   1.2.1      v stringr 1.4.1
## v readr   2.1.3      v forcats 0.5.2
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

The `open_dataset` function connects us to the online parquet file (gbif snapshot from above). Notice that this doesn't load the entire dataset into your computer's memory. Conveniently, we can still take a look at the variable names (and even take a glimpse at a small subset of the data)

```
db <- open_dataset(gbif_snapshot)
db # take a look at the variables we can query
glimpse(db) # see what a subset of the data looks like
```

Step 2. Querying the data

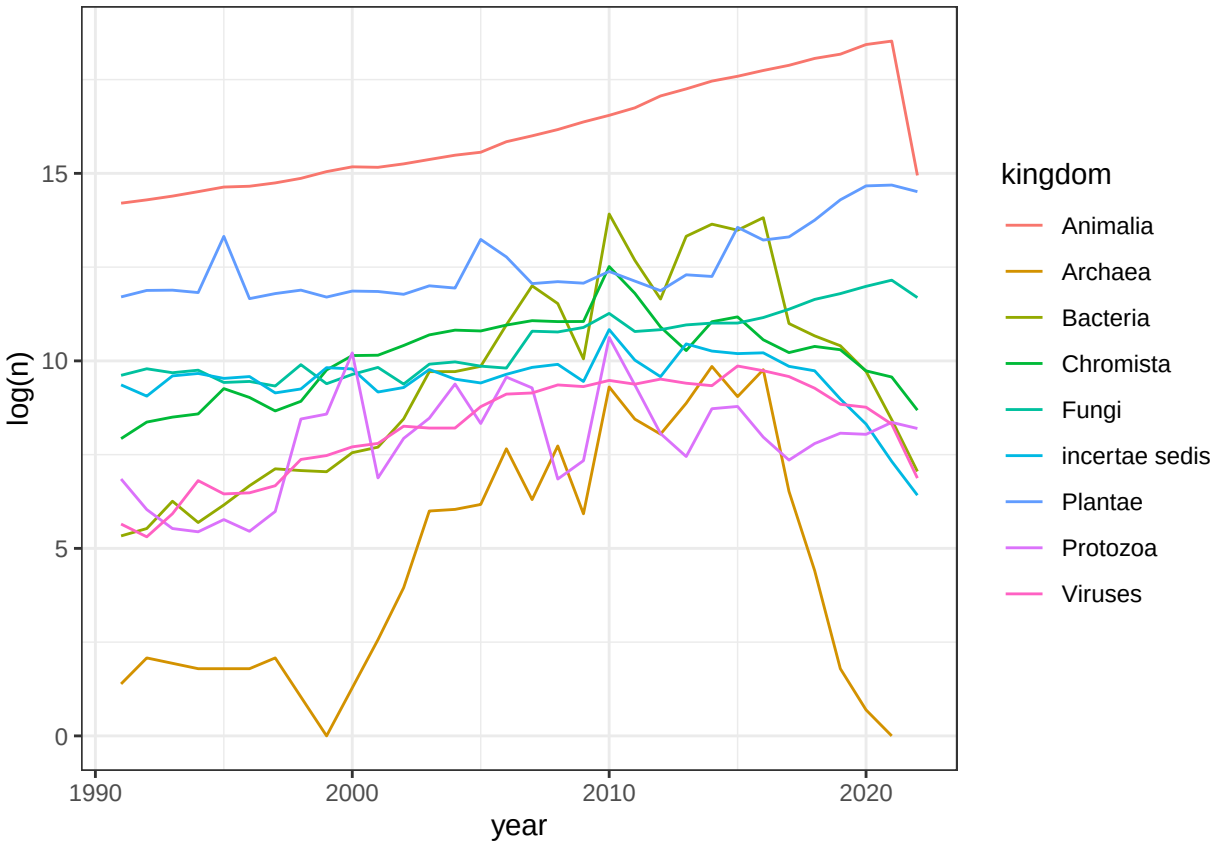
The arrow R package provides a dplyr interface to Arrow Datasets - this is what we will use to perform the query. Note that the this a bit limited – only a subset of dplyr verbs are available to query arrow dataset objects (there is nice resource here). But verbs like `filter`, `group_by`, `summarise`, `select` are all supported.

So lets filter GBIF to all observation in the United States and get a count of observations of each class of species per year. We use the `collect()` function to pull the result of our query into our R session.

```
gbif_US <- db |>
  filter(countrycode == "US") |>
  group_by(kingdom, year) |>
  count() |>
  collect()
```

Within minutes we can plot the data we collected through time!

```
gbif_US |>
  drop_na() |>
  filter(year > 1990) |>
  ggplot(aes(x = year,
             y = log(n),
             color = kingdom)) +
  geom_line() + theme_bw()
```



There are lots of other applications of arrow that we won't get into here (e.g., zero-copy R and python data sharing). Check out the "cheat sheet" linked here if you are interested in exploring!

Raster data: STAC and vsicurl

Cloud native workflows are not limited to querying tabular data (e.g., parquet, csv files) but can also be super helpful for working with spatiotemporal data. We'll focus on reading and querying data using SpatioTemporal Asset Catalog (STAC) (STAC is just a common language to describe geospatial information, so it can more

easily be worked with, indexed, and discovered. More info here). Lots of spatiotemporal data is in STAC – for example, Microsoft planetary computer data catalog includes petabytes of environmental monitoring data using STAC format.

We'll use the `rstac` library to make requests to the planetary computer's STAC API (and like our tabular example, similar workflows are available in python)

```
library(tidyverse)
library(rstac)
library(terra)
library(sf)
```

```
## Search for data from a given collection in a given bounding box:
s_obj <- stac("https://planetarycomputer.microsoft.com/api/stac/v1/")
```

We can set a bounding box for our data search - lets say, the SF bay area

```
SF <- st_read("https://dsl.richmond.edu/panorama/redlining/static/downloads/geojson/CASanFrancisco1937.g
```

```
## Reading layer 'CASanFrancisco1937' from data source
##   'https://dsl.richmond.edu/panorama/redlining/static/downloads/geojson/CASanFrancisco1937.geojson'
##   using driver 'GeoJSON'
## Simple feature collection with 97 features and 4 fields
## Geometry type: MULTIPOLYGON
## Dimension:     XY
## Bounding box:  xmin: -122.5101 ymin: 37.70801 xmax: -122.3627 ymax: 37.80668
## Geodetic CRS:  WGS 84
```

```
bbox = st_bbox(SF)
```

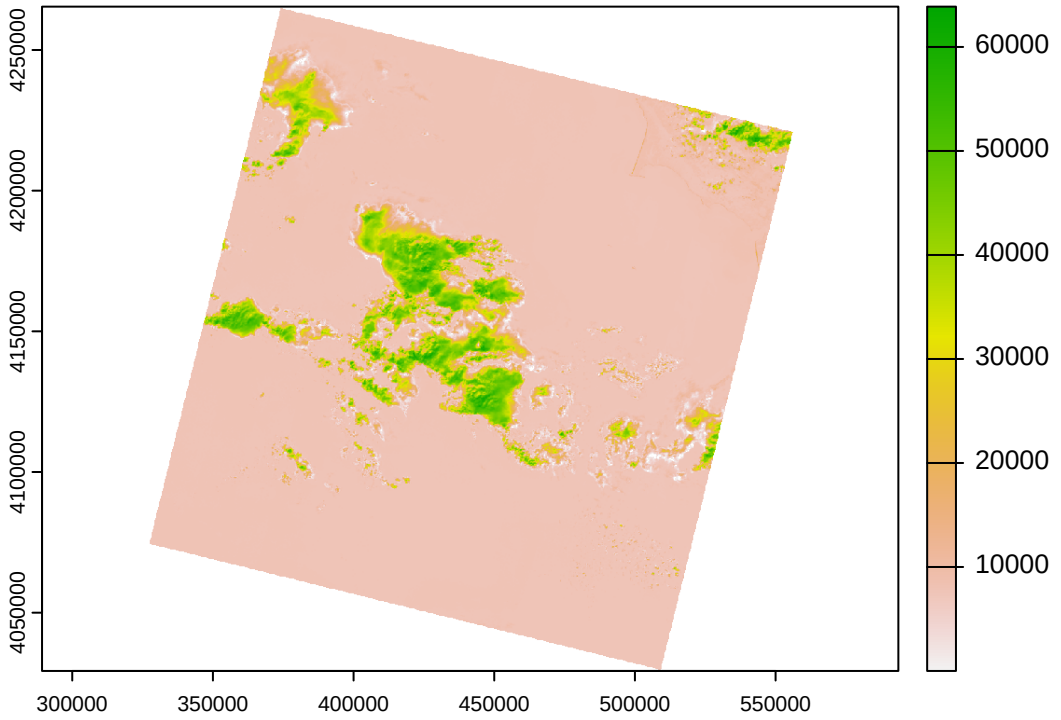
And we can take a look at a given collection (e.g., Landsat data)

```
it_obj <- s_obj %>%
    stac_search(collections = "landsat-c2-l2",
                bbox = bbox) %>%
    get_request() |>
    items_sign(sign_fn = sign_planetary_computer())
```

```
## Warning: Items matched not provided.
```

Instead of downloading the raster, we use the prefix `/vsicurl/` and the download URL from above - passing that directly to the `rast()` function (reads in a raster; from the `terra` package)

```
url <- paste0("/vsicurl/", it_obj$features[[1]]$assets$blue$href)
data <- rast(url)
plot(data)
```



We can do computation over the network too - for example calculating NDVI using a subset of bands in Landsat of Sentinel data

[add a bit more text about what you can do and point to some more examples..]