neonstore: provenance-aware persistent storage for NEON data

Carl Boettiger[a], R. Quinn Thomas[c], Christine M. Laney[b], Eric Sokol[b], Claire Lunch[b]

[a]*Dept of Environmental Science Policy and Management University of California Berkeley Berkeley CA 94720-3114 USA*
[b]*NEON*
[c]*Virginia Tech*

Abstract

The National Ecological Observatory Network

The National Ecological Observatory Network (NEON) represents an important resource and a major investment in data collection infrastructure needed to address many of the grand challenges facing environmental science [5, 3, 2]. NEON provides access to 181 (active and planned) ecological data products at 81 (permanent and relocatable) sites throughout the United States. The `neonstore` R package seeks to provide provenance aware, high performance access and persistent storage of NEON data files. Here, we describe the design, use, and rationale for the package. We illustrate a few example use cases, including working with larger-than-RAM data and cloud-native access patterns that avoid direct downloads. The approaches taken in `neonstore` maximize reproducibility, facilitate collaboration including across computer languages, and help making analyses which leverage the full spatial and temporal scale of NEON easily available to researchers with modest computational resources and expertise.

Provenance and performance are two central aspects to the design of `neonstore`. A focus on provenance means that analyses can always be traced back to individual NEON data files, and supports both workflows that require access to the most recent or updated data and workflows that require repeated access to persistent, stable data files. A focus on performance reflects the potentially very large size of its aggregated data products, which can easily exceed available working memory (RAM) on most machines: as many unified tables of even single products over all available sites and dates can range from 10 GBs into the terrabyte range. Because enabling analyses that could span the full spatial and temporal range of NEON's 81 sites over a 30 year horizon is a fundamental motivation for NEON research, it essential that researchers be able to easily work with data across this full span. However, doing so requires techniques that go beyond workflows which expect data to be re-downloaded (provenance) or stored in memory for analysis (performance).

To address these challenges, `neonstore` implements a high-performance, provenance aware workflow by drawing on a host of key technologies, including cryptographic hashes, a Lightning Memory Mapped Database (LMDB) for metadata management, a columnar-oriented integrated relational database (duckdb) for data management, and support for remote access through cloud-native object stores. These technologies are seamlessly integrated behind a user-friendly R interface which requires no expertise in the underlying backend technology beyond a familiarity with R and the "tidy data" model used in the popular R packages, `dplyr` and `tidyr`.

Performance

`neonstore` is also concerned with performance of research workflows based on NEON data. Native operations in R are based in memory, making it difficult to work with objects that are larger than RAM. Researchers seeking to take full advantage of NEON data collected across all available sites and years can quickly run ub against the limits of data which can be held and processed in working memory. The R language also supports interfaces with relational databases [6], with the widely used package `dplyr` capable of

*Corresponding author

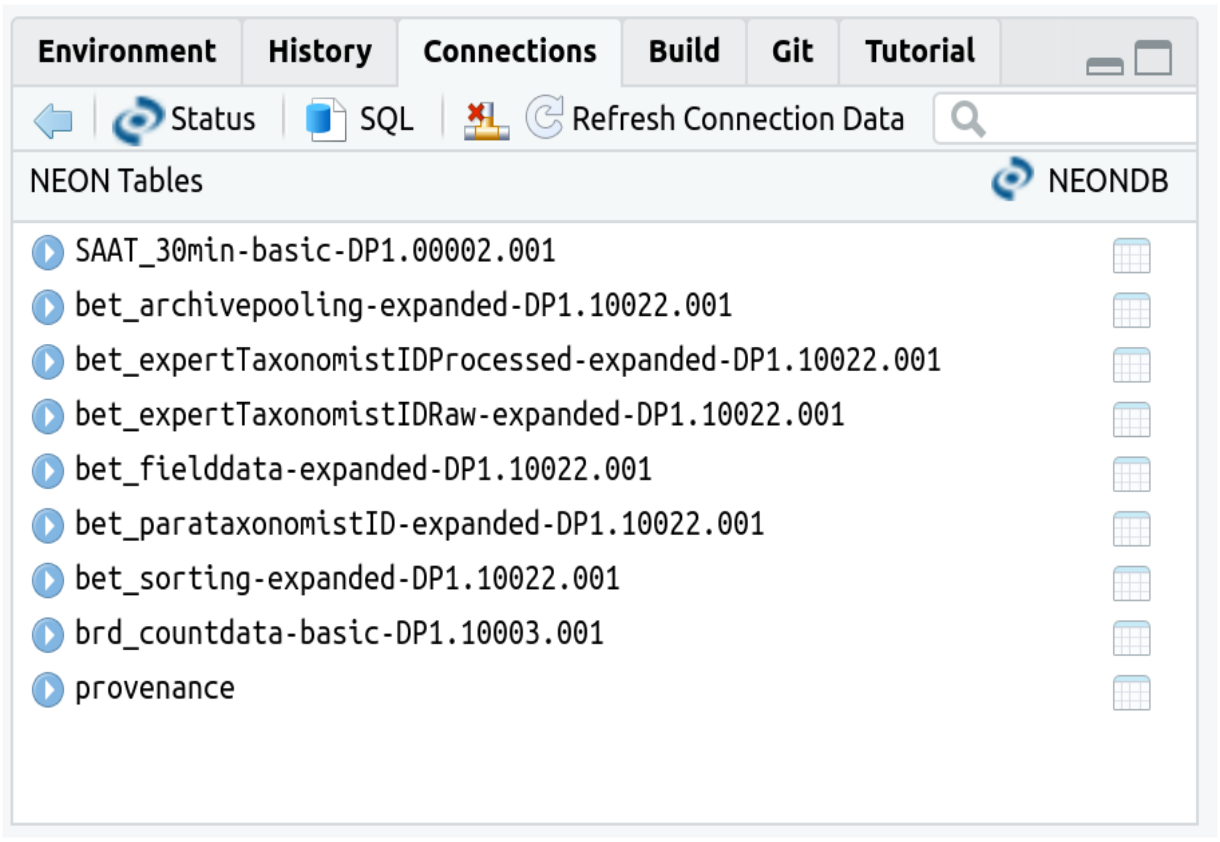
seamlessly tanslating common functions (e.g. `filter`, `select`, `mutate`, `group_by`, `summarise`, `inner_join`, etc) into SQL queries that can be run by external database servers [8, 9]. This allows users to perform common tasks on data much larger than would fit into working RAM. Traditional relational databases such as MariaDB and Postgres use a server-client model with row-oriented data structures, suitable for concurrent read and write operations by many simultaneous client processes acccessing a central database (e.g. many client banks simultaneously updating a central registry of accounts, see ACID). Modern data science workflows frequently benefit more from column-oriented data structures where simultaneous operations (ACID transactions) are not required. This makes it possible to both eliminate the requirement of separately installing and running a database server while also opening the door to substantial performance improvements beyond the reach of traditional relational database clients. `duckdb` is a leading example of such a columnar database [7, 4], with seamless support for R integration and superior performance benchmarks to both traditional relational databases and in-memory operations in R using `dplyr`.

`neonstore` allows users to easily import tabular NEON data products (including `.h5` based eddy covariance products) into a local `duckdb` database using the function `neon_store()`. If desired, users can instead use their own existing databases by providing the appropriate `DBI::dbConnection` object to the `db` argument of `neon_store()`. `neon_store()` stacks corresponding individual raw tabular data files by month, site, and (for IS data) sensor position into individual database tables. `neon_store()` is capable of detecting revisions to previously released data products, removing outdated rows and replacing them with updated rows without accidental duplication of data. Data revisions are indicated by downloading files which share the same data product, site, month and sensor position file name as files already present in the local store, but have an updated timestamp and differing content hash. `neon_store()` provides a warning whenever existing data tables are updated, as this may require previous analyses to be re-run. Because sensor data does not indicate the site or position information in individual data files (this is provided only through the file name) `neonstore` stacking functions add additional columns for these variables automatically. Additionally, a `file` column is always included indicating the raw file source for each row. Many NEON data products come in two types: a "basic" format and an "expanded" format which may include additional columns providing data fields that were not included in the original product specification. NEON does not always use globally unique names for table descriptions, particularly for metadata tables such as `sensor_positions`. To avoid ambiguities created by these issues, database table names are formed using the table description name, the product type, and the product number, e.g. `brd_countdata-basic-DP1.10003.001`, rather than the description component alone, `brd_countdata`. RStudio users can bring up an interactive list of available tables in the RStudio "Connections" pane using the function `neon_pane()` ??.

`neonstore` seeks to help users download and manage NEON files locally in a way that lets analyses be quickly traced back to the raw data files. `neonstore` is distinguished as much by what it does not do as by what it does. All `neonstore` operations are intended to be transparent and intuitive, and could easily be duplicated in another computational language or command line interface. Specifically, `neonstore` will never automatically "clean" data: opaque data transformations make output sensitive to the version and implementation details of the tool in question. A user should expect the same file from a "download" operation regardless of the software or software version used to perform the download. Functions which in read in the data should faithfully reproduce the raw data so that analyses can easily be compared to those using other software tools. When functions perform tasks like guessing the data type (numeric, string, Date, etc) of a column, or adding additional columns to stack two otherwise matching tables in which one has recorded an additional variable, they should do so in a manner that is transparent, controllable, and consistent with the behavior of common functions used in the language.

`neonstore` Functionality

The `neonstore` package can be divided into two parts. One part works directly with the NEON API to handle tasks of *data discovery* and *data download* into a persistent local store, while the other provides three mechanisms to work with data.



SAAT_30min-basic-DP1.00002.001
bet_archivepooling-expanded-DP1.10022.001
bet_expertTaxonomistIDProcessed-expanded-DP1.10022.001
bet_expertTaxonomistIDRaw-expanded-DP1.10022.001
bet_fielddata-expanded-DP1.10022.001
bet_parataxonomistID-expanded-DP1.10022.001
bet_sorting-expanded-DP1.10022.001
brd_countdata-basic-DP1.10003.001
provenance

Figure 1: RStudio Connections pane showing interactive panel listing all tables imported into the local duckdb database.

```r
library(neonstore)
library(dplyr)
```

Data discovery is facilitated by functions which query the NEON API for information about data contained in it's catalogue. The function `neon_products()` returns a data frame with a description and important metdata about each product, including the variables measured and the sites and dates which it covers.

```r
neon_products()
#> # A tibble: 182 x 25
#>    productCodeLong            productCode   productCodePresentation productName
#>    <chr>                      <chr>         <chr>                   <chr>
#>  1 NEON.DOM.SITE.DP1.00001.001 DP1.00001.001 NEON.DP1.00001          2D wind sp~
#>  2 NEON.DOM.SITE.DP1.00002.001 DP1.00002.001 NEON.DP1.00002          Single asp~
#>  3 NEON.DOM.SITE.DP1.00003.001 DP1.00003.001 NEON.DP1.00003          Triple asp~
#>  4 NEON.DOM.SITE.DP1.00004.001 DP1.00004.001 NEON.DP1.00004          Barometric~
#>  5 NEON.DOM.SITE.DP1.00005.001 DP1.00005.001 NEON.DP1.00005          IR biologi~
#>  6 NEON.DOM.SITE.DP1.00006.001 DP1.00006.001 NEON.DP1.00006          Precipitat~
#>  7 NEON.DOM.SITE.DP1.00007.001 DP1.00007.001 NEON.DP1.00007          3D wind sp~
#>  8 NEON.DOM.SITE.DP1.00010.001 DP1.00010.001 NEON.DP1.00010          3D wind at~
#>  9 NEON.DOM.SITE.DP1.00013.001 DP1.00013.001 NEON.DP1.00013          Wet deposi~
#> 10 NEON.DOM.SITE.DP1.00014.001 DP1.00014.001 NEON.DP1.00014          Shortwave ~
#> # i 172 more rows
#> # i 21 more variables: productDescription <chr>, productStatus <chr>,
#> #   productCategory <chr>, productHasExpanded <lgl>,
#> #   productScienceTeamAbbr <chr>, productScienceTeam <chr>,
#> #   productPublicationFormatType <chr>, productAbstract <chr>,
#> #   productDesignDescription <chr>, productStudyDescription <chr>,
#> #   productBasicDescription <chr>, productExpandedDescription <chr>, ...
```

Similarly, metadata about all sites is avialable from `neon_sites()`.

Data downloading is handled by the `neon_download()` function, which can take a vector of sites or products and date ranges, and can download specific tables or all tables and metadata included in a given product. The function handles rate limiting and the use of NEON tokens, see appendix or package README for more details including comparison to functionality in `neonUtilities`. Downloaded data is stored in a persistent, configurable location, and `neonstore`'s provenance tracking system avoids re-downloading files that were previously downloaded, and ensures that updated or corrected data is always downloaded without overwriting earlier versions.

```r
neon_download("DP1.10003.001")
```

Note that if we immediately attempt to re-download a subset of this data, `neonstore` detects that we have precisely matching files already downloaded to our local store:

```r
neon_download("DP1.10003.001",
              start_date = "2018-01-01",
              end_date = "2019-01-01",
              site = "YELL")
```

Interacting with downloaded files

`neonstore` provides three mechanisms for working with downloaded data, each of which builds on the former: a file interface, a relational database, and remote database. The relational database helps `neonstore` scale to big data analyses, while the remote database mechanism extends that further to scale to research teams.

File-based Interface: `neon_index()` *&* `neon_read()`

Even adding a handful of products can generate thousands or hundreds of thousands of files in a local store. Surprisingly, these can even exceed the capacity of common `bash` shell utilities like `ls`. `neonstore` sidesteps this issue through a convenient R-based interface. The function `neon_index()` will list all recognized files found in the local store as a `data.frame`. `neon_index()` uses a sophisticated filename parser, `neon_parse_filenames()`, that understands all documented file naming conventions listed at `https://data.neonscience.org/file-naming-conventions`, and quite a few that are not yet listed (internal NEON communication). The most common of these are displayed in the table:

```
neon_index()
#> # A tibble: 1,217 x 15
#>    product  site  table type  ext   month timestamp           horizontalPosition
#>    <chr>    <chr> <chr> <chr> <chr> <chr> <dttm>                           <dbl>
#>  1 DP1.100~ BART  brd_~ basic csv   2015~ 2022-11-22 18:06:13                 NA
#>  2 DP1.100~ BART  brd_~ basic csv   2016~ 2022-11-22 18:28:29                 NA
#>  3 DP1.100~ BART  brd_~ basic csv   2017~ 2022-11-22 18:51:55                 NA
#>  4 DP1.100~ BART  brd_~ basic csv   2018~ 2022-11-28 18:02:03                 NA
#>  5 DP1.100~ BART  brd_~ basic csv   2019~ 2022-11-28 18:54:56                 NA
#>  6 DP1.100~ BART  brd_~ basic csv   2020~ 2022-11-28 21:00:18                 NA
#>  7 DP1.100~ BART  brd_~ basic csv   2020~ 2022-11-28 21:57:32                 NA
#>  8 DP1.100~ BART  brd_~ basic csv   2021~ 2022-11-29 23:48:16                 NA
#>  9 DP1.100~ BART  brd_~ basic csv   2022~ 2023-12-29 05:32:56                 NA
#> 10 DP1.100~ BART  brd_~ basic csv   2015~ 2022-11-22 18:06:13                 NA
#> # i 1,207 more rows
#> # i 7 more variables: verticalPosition <dbl>, samplingInterval <chr>,
#> #   date_range <chr>, path <chr>, md5 <chr>, crc32 <chr>, release <chr>
```

Note that the index also contains important provenance information, including the checksum (MD5 or CRC32), timestamp, and release tag indicated by NEON. Internally, `neonstore` uses checksums to track the provenance of all files downloaded, which allows it to handle both the cases when NEON changes the name of a file but not the contents, and when NEON changes the contents of a file, even if the name remains unchanged. `neon_index()` takes optional arguments such as `product`, `start_date`, `end_date`, and `site` to allow for convenient sub-setting.

`neon_index()` also allows users to access metadata files, such as README and EML files associated with each data product and/or NEON site.

Stacking datasets

The highly atomized file structure breaks tables down by site and year/month as individual csv files (and other formats, including h5). Typical workflows will treat a collection of such files as a single data table, with columns indicating the site, year, and month of sampling. When combining the collection of files into a single table (a process sometimes known as stacking), `neonstore` must handle several special cases:

1) Not all files have identical columns in identical order across all sites and months. This can break simple, high-performance file readers like `vroom` and `arrow`.
2) The data types of columns (string, numeric, date, timestamp, etc) must be inferred correctly. This can be difficult when certain columns within individual files have no non-missing (`NA`) values.
3) Additional columns must be added to capture information (metadata) that is only available in the file name, such as siteID and position (for sensor-based products).
4) Most importantly, reading must be provenance-aware – if there are two files with matching product, table, site, year-month and position (for sensor-based products) must resolve which file should be read in based on the timestamp provided, or the requested version.

`neonstore`'s stacking functions handle all of this for us. When a version of the same data is detected with a revised timestamp compared to an earlier download, `neonstore` will notify us of the detection, but default to reading the most recent available version.

`neonstore` provides two alternative mechanisms for stacking data. The simplest approach is `neon_read()`, which stacks all matching files into a single, in-memory `data.frame`. Note that `neon_read()` can be given arguments to filter by site, date range, etc to read in a particular selection of data more quickly. `neon_read()` always reads directly from the original, raw data files downloaded to the local neonstore.

```
brd_countdata <- neon_read("brd_countdata-expanded")
```

Relational Database Interface

For very large tables such as sensor data collections spanning many sites and years, users will require the increased performance provided by the high-performance, columnar-oriented relational database, `duckdb` [7, 4]. To take advantage of this approach, we must first store the contents of the individual data files in the relational database using the function `neon_store()`:

```
neon_store("brd_countdata")
```

This task needs to be done only once for each table or product after any new files have been downloaded. Thanks to provenance tracking, any repeated calls to this function will not result in additional imports. If a previously imported file has been deprecated by the publication of a matching file with newer timestamp, `neon_store()` will remove the deprecated data and import the new data, with a warning saying so.

Once data has been imported into the local database, we can quickly load the table into R with `neon_table()`

```
brd_countdata <- neon_table("brd_countdata")
```

This will be somewhat faster than `neon_read()`, since there is no need to parse the original data. By default, this will still read all the data into a normal in-memory `data.frame` in R. When working with very large data tables, it may be preferable or necessary to work with data directly in duckdb, which can perform most `dplyr` and `tidyr` operations, such as `group_by()`, `summarise`, `filter`, and so forth, within the database.

```
brd_countdata <- neon_table("brd_countdata", lazy = TRUE)
brd_countdata
#> # Source:   table<brd_countdata-basic-DP1.10003.001> [?? x 24]
#> # Database: DuckDB v0.9.2 [unknown@Linux 6.5.6-76060506-generic:R 4.3.1//home/cboettig/.local/share
#>    uid                         namedLocation domainID siteID plotID plotType pointID
#>    <chr>                       <chr>         <chr>    <chr>  <chr>  <chr>    <chr>
#>  1 f7fa2f5a-5b07-4ac0-83b~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  2 84c1e17a-945d-46fa-a1f~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  3 4063e302-4b9a-45ff-9a6~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  4 53e2c631-d1e1-4156-b1f~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  5 51cdba5c-64a9-4abf-aff~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  6 d742982a-1052-4d3f-bb6~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  7 2c86f910-5cba-4dc0-adf~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  8 dbf436ae-89af-46ac-980~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#>  9 da7d0c2a-6d06-4748-a21~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#> 10 23938ad7-76fc-4e48-a67~ TREE_022.bas~ D05      TREE   TREE_~ distrib~ 21
#> # i more rows
#> # i 17 more variables: startDate <dttm>, eventID <chr>, pointCountMinute <dbl>,
#> #   targetTaxaPresent <chr>, taxonID <chr>, scientificName <chr>,
#> #   taxonRank <chr>, vernacularName <chr>, observerDistance <dbl>,
#> #   detectionMethod <chr>, visualConfirmation <chr>, sexOrAge <chr>,
#> #   clusterSize <dbl>, clusterCode <chr>, identifiedBy <chr>,
#> #   identificationHistoryID <chr>, file <chr>
```

Note that R displays what appears to be the same data frame as before, but with two essential differences. R indicates that the displayed object is a `duckdb_connection`, and lists the number of rows only as `??` (or within RStudio's rmarkdown interface, only the first 1000 rows will be displayed in the interactive table). The number of rows is not displayed because this query is executed "lazily." Since the print method needs only display the first 6 rows, the database does not execute the potentially expensive operation of counting all the rows in the table, it merely displays the first 6. This approach is particularly pwoerful if we wish to first subset or summarise the data in a way that will result in an overall smaller table.

Consider the example of a much larger table collected by sensor data; say, single-aspirated temperature collected every 30 minutes across all sites. We download and import the 30 minute sampled table (ote that we exploit pattern matching to provide a short-hand to the table name). The initial download is not a fast operation - in our example, rate-limiting rules by NEON make the command take over 20 minutes to complete the API requests that would otherwise take only 3 minutes. The download itself is determined more by network speed. In our example this takes about 1 hour to download over 13 thousand individual files.

`neon_store()` import is also a slow, one-time operation to read in all of these files into the database, depending on disk and precessor speeds. In our example, this takes about 20 minutes. Note that it is not necessary to repeat these commands later except to import newly released data, which will be added to the persistent store.

```
neon_download(product = "DP1.00002.001", table = "30min") #Temp single aspirated
neon_store(product = "DP1.00002.001", table = "30min") #Temp single aspirated
```

With the data imported into the database, `neon_table()` can quickly connect to the remote database.

```
temp <- neon_table(product = "DP1.00002.001",  table = "30min", lazy=TRUE) #Temp single aspirated
```

This is a lot of data! Over 19 million rows:

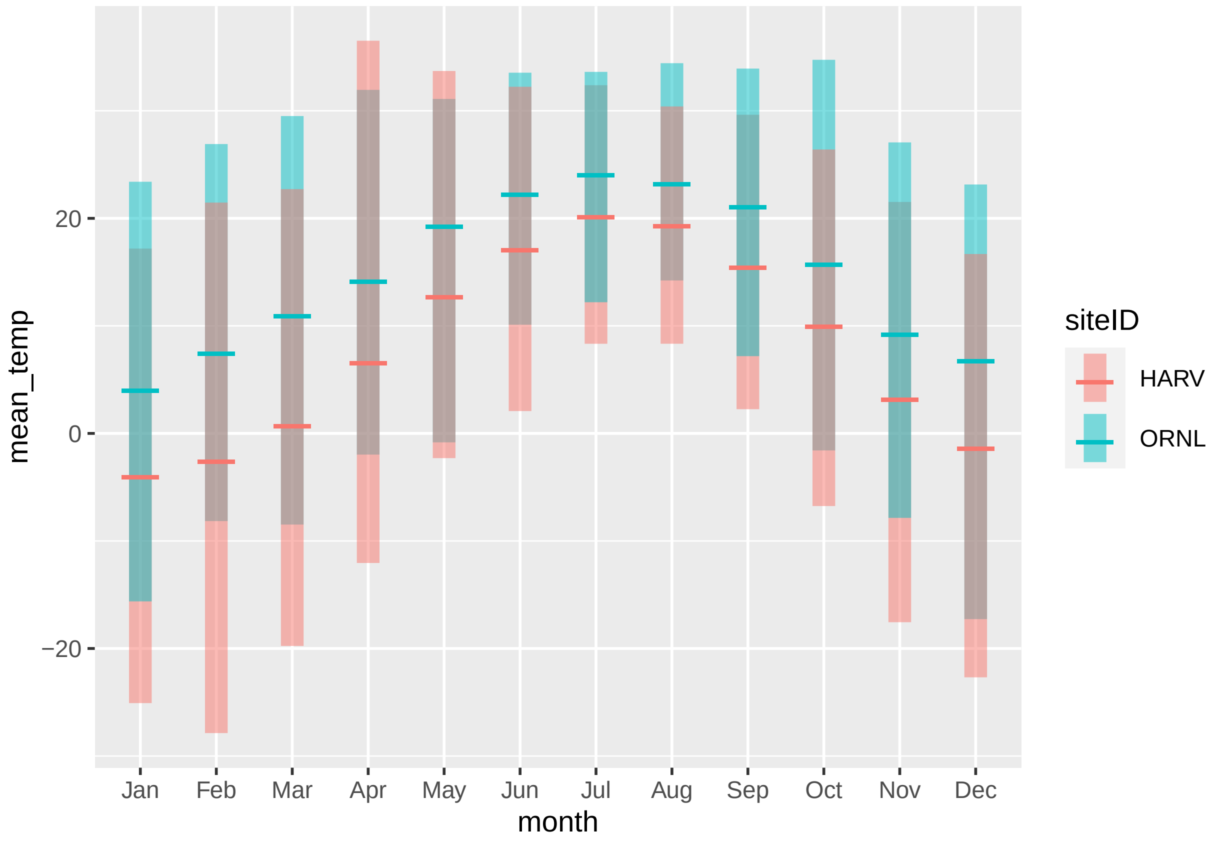
```
temp %>% count()
#> # Source:   SQL [1 x 1]
#> # Database: DuckDB v0.9.2 [unknown@Linux 6.5.6-76060506-generic:R 4.3.1//home/cboettig/.local/shar
#>          n
#>      <dbl>
#> 1 21062784
```

However, working with the remote duckdb connection, we can compute summary statistics, such as monthly averages for each site, takes only milliseconds:

```
bench::mark({
temp_monthly <- temp %>%
  mutate(month = month(startDateTime)) %>%
  group_by(siteID, month) %>%
  summarise(mean_temp = mean(tempSingleMean, na.rm=TRUE),
            max = max(tempSingleMean, na.rm=TRUE),
            min = min(tempSingleMean, na.rm=TRUE),
            .groups = "drop") %>%
  collect()
})
#> # A tibble: 1 x 6
#>   expression                                 min median `itr/sec` mem_alloc `gc/sec`
#>   <bch:expr>                               <bch> <bch:>     <dbl> <bch:byt>    <dbl>
#> 1 { temp_monthly <- temp %>% mutate(m~ 167ms  171ms      5.83    3.01MB     2.92
```

Not only is this about 10x faster than if we had read the table entirely into R first, but this also executes with minimal memory footprint (317Kb) instead of the over 2.6 GB required to perform the same operation in R. Having summarized the data using our desired statistics, the `collect()` command has returned the result as a regular in-memory `data.frame` which we can use with other R functions, such as data visualization tools:

```
library(ggplot2)
library(lubridate)
temp_monthly %>% mutate(month = lubridate::month(month, label=TRUE)) %>%
  filter(siteID %in% c("HARV", "ORNL")) %>%
  ggplot(aes(month, mean_temp, col=siteID, ymin = min, ymax = max, fill = siteID)) +
  geom_linerange(lwd=4, alpha=.5) +
  geom_point(size=8, shape="_")
```



Similarly, it is straight forward to use most other `dplyr` operations, such as the joins, to combine data across multiple tables. The promise of NEON is precisely this ability to work across a large spatial and temporal range of data collected in standardized fashion.

Cloud-based access

Cloud-based access offers a simpler and sometimes faster alternative to the standard pattern of first downloading the data (with `neon_download()`), importing to the local database (with `neon_store()`) and then accessing from the database (with `neon_table()`). In cloud-based access, we simply access data directly from the Google Cloud Storage platform without downloading:

```
df <- neon_cloud(product = "DP1.00002.001", table = "30min") # Temp single aspirated
```

Unfortunately, at this time NEON still requires us to access the API to obtain a list of individual file URLs. The API is quite slow, because that list of files requires many separate API requests (for each site and for each year-month in the request). In future this may become a single request and will thus be almost instantaneous.

`neon_cloud()` does not immediately download any of the actual data fields. Instead, it merely reads the metadata (the table schema, i.e. column names) of the first CSV in the collection. This approach is often referred to as "lazy evaluation" in software engineering design – the function executes as quickly as possible by deferring any computation until it is absolutely necessary. For instance, if we are going to work with only a few columns of the data, we can save computational effort by not downloading and parsing all those extra columns. Despite this strategy, our `df` object "feels" like a normal data.frame in most ways – we can perform most of the usual operations as if the data were already both downloaded and read into memory. Here we immediately perform the same calculation as before:

```
temp_monthly <- df %>%
  mutate(month = month(startDateTime)) %>%
  group_by(siteID, month) %>%
  summarise(mean_temp = mean(tempSingleMean, na.rm=TRUE),
            max = max(tempSingleMean, na.rm=TRUE),
            min = min(tempSingleMean, na.rm=TRUE),
            .groups = "drop") %>%
  collect()
```

Provenance

`neonstore` focus on provenance emphasizes being able to trace results back precisely to individual raw data files provided by the NEON API. NEON data products are distributed using a highly atomized file design: with each data product from either both observation systems (OS) and instrument systems (IS) divided into separate files by site and month. Products generated by the airborne observation platform (AOP) are typically divided by type, flight date, flight line, and grid cell. This atomized approach facilitates sub-setting and data transfer, because users can identify ahead of time which products, locations, and time ranges they need. As some NEON data products can be quite large (NEON's complete data catalog to date is already in excess of 1 petabyte, primarily due to the size of many AOP products), being able to download or read only the files of interest is already important to save bandwidth or computer memory, or disk space. This approach also facilitates data provenance and reproducibility, because it means that an update or correction to data collected at a particular site and time results only in a change to a single file and not the whole data product. However, the highly atomized structure of NEON's data can also become a barrier to entry. Most analyses will inevitably require multiple files to be combined or stacked, e.g. to create a unified record for a product across many months or years of sampling or across many sites. `neonstore` provides functions to stack data products that have been atomized into separate files by month, site or sensor position into single `data.frame` objects, store stacked tables in a local, high-performance database, and track individual data rows back to their raw data files.

Managing provenance of NEON data means dealing with the addition of new data, corrections published to old data, and changes to file names that do not represent changes in underlying data. Any analysis is only as good as as the data supporting it. Because NEON data is subject to both revisions (where previously released data are corrected) and new releases (where more recently collected data is made public), researchers will seek to update previous analyses with corrected data and new observations. `neonstore` enables scripted workflows through which such updates can occur automatically by re-running the same script. More confusingly, NEON frequently updates the timestamp associated with a file even when the file has not changed. Typically these events are associated with the update of a separate but related file. In such cases, `neonstore` can avoid downloading and importing these unchanged data, ensuring that automated workflows do not run unnecessarily. When new or revised data significantly alter results, it is also important to be able to pinpoint precisely what has changed and why. These elements creates five distinct tasks for managing NEON data provenance: (1) The ability to determine which of each of the individual raw data files were used in an analysis (2) The ability to append new data as it is collected and released (3) the ability to transparently replace existing data when corrections are published, (4) the ability to avoid re-downloading or re-stacking data that has been previously processed, (5) the ability to freeze the data used in a given analysis, so that results can reliably be repeated without accidentally including revised or newer data products.

To provide these five abilities for data provenance, `neonstore` maintains a local file store containing all raw data files downloaded from NEON. Additionally, each file is entered into a local registry, along with the file's MD5 checksum (CRC32 for AOP products) and currently associated NEON release (if any). This local registry is kept in a Lightning Memory Mapped Database, LMDB, an extremely fast, open source key-value store indexed on both file name and checksum. MD5 checksums effectively indicate unique content. It is theoretically possible for a malicious actor to create a file with different contents but the same checksum, but vanishingly unlikely to occur by chance. On requesting new files to download, `neonstore` compares file names and hashes reported by the API to those in the local store, allowing it to omit any data that has already been downloaded, even if the timestamp in a file name has changed but the contents remain the same.

NEON's filenaming convention provided essential metadata about each file, such as the data product (product level, product number, revision number), site, year and month of the release, file description, a timestamp. Sensor data (IS) also include additional information about the horizontal and vertical position and meaasurement interval (frequency) of the sensor product. AOP products use a somewhat different convention with flight information. `neonstore` recognizes all published NEON file naming conventions and automatically parses NEON filenames into their component parts. Data are organized into subdirectories based on product, site, and year-month respectively. Data or metadata files shared across all months or all sites of a product are stored in the corresponding parent directory. The function `neon_index()` returns a `data.frame` listing every recognized individual file in the local store, along with the corresponding metadata fields parsed from the filename. Optional arguments can be given to restrict this table to files matching specific products, sites, date ranges, etc.

Conclusions

`neonstore` provides a high-performance and provenance-aware mechanism for accessing, maintaining, and working with NEON data products, designed to scale seamlessly to analyses which stretch across the full spatial and temporal range which makes this nation-wide dataset so uniquely valuable. The `neonstore` package follows simple and robust best-practices for data management that can work across most popular computing languages; such as preserving a local library of the original raw data files downloaded from NEON, and importing data into a local relational database (`duckdb`) that can be accessed from a wide range of libraries. Additional utilities enable serverless network-based sharing of a large `neonstore` database across a team through the use of `parquet` and `arrow`'s remote filesystems (which likewise supports most major computing languages.) `neonstore` has been co-developed by NEON staff (CML & CL) and members of the Ecological Forecasting Initiative (CB & RQT), tested in supporting the EFI NEON forecasting challenge [1], among other use cases.

References

[1] NEON ecological forecasting challenge. URL `https://projects.ecoforecast.org/neon4cast-docs/`.

[2] Harnessing the NEON data revolution to advance open environmental science with a diverse and data-capable community. *Ecosphere*, 12(12), December 2021. ISSN 2150-8925, 2150-8925. doi: 10.1002/ecs2.3833. URL `https://onlinelibrary.wiley.com/doi/10.1002/ecs2.3833`.

[3] Jennifer K Balch, R Chelsea Nagy, and Benjamin S Halpern. NEON is seeding the next revolution in ecology. *Frontiers in Ecology and the Environment*, 18(1):3–3, February 2020. ISSN 1540-9295, 1540-9309. doi: 10.1002/fee.2152. URL `https://onlinelibrary.wiley.com/doi/10.1002/fee.2152`.

[4] Hannes Mühleisen, Mark Raasveldt, and DuckDB Contributors. *duckdb: DBI Package for the DuckDB Database Management System*, 2021. https://duckdb.org/, https://github.com/duckdb/duckdb.

[5] National Research Council. *NEON: Addressing the Nation's Environmental Challenges*. The National Academies Press, Washington, DC, 2004. ISBN 978-0-309-09078-0. doi: 10.17226/10807. URL `https://www.nap.edu/catalog/10807/neon-addressing-the-nations-environmental-challenges`.

[6] R Special Interest Group on Databases (R-SIG-DB), Hadley Wickham, and Kirill Müller. *DBI: R Database Interface*, 2021. https://dbi.r-dbi.org, https://github.com/r-dbi/DBI.

[7] Mark Raasveldt and Hannes Mühleisen. DuckDB. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, jun 2019. doi: 10.1145/3299869.3320212. URL `https://doi.org/10.1145/3299869.3320212`.

[8] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. *dplyr: A Grammar of Data Manipulation*, 2021. https://dplyr.tidyverse.org, https://github.com/tidyverse/dplyr.

[9] Hadley Wickham, Maximilian Girlich, and Edgar Ruiz. *dbplyr: A 'dplyr' Back End for Databases*, 2021. https://dbplyr.tidyverse.org/, https://github.com/tidyverse/dbplyr.