

Using TileDB with R

An Introductory Tutorial



7 July 2021 @ useR! 2021



Overview

Outline

Brief Introduction

Key Topics

- Dense Arrays
- Sparse Arrays
- Full TileDB API
- S3 Access
- Arrow Format
- Time Travel
- Encryption

Applications

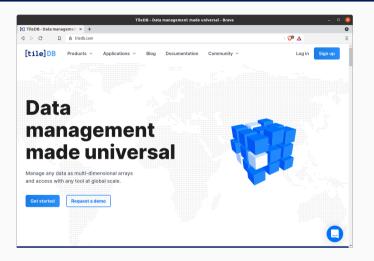
- SQL Access Example
- Data Science with Flights
- LiDAR / Geospatial
- Finance / Time Series
- Genomics: GWAS

Wrap-Up

Further References

Introduction

TileDB



Universal Data Management

Any Data in Multi-Dimensional Arrays

Serverless, and at scale

In this tutorial with an R focus



Tutorial Resources

To install the package with code examples and the slides, use

```
remotes::install_github('eddelbuettel/tiledb-user2021')
```

or

Loading the package will show where the example files are located.

The conference slack channel for the tutorial is #tut_tiledb.

Introductory Example

[tile]DB

```
# if needed: install.packages("tiledb")
                                             # installation from CRAN
library(tiledb)
                                             # load the package
library(palmerpenguins)
                                             # example data
setwd("/tmp")
                                             # or other scratch space
# create array from data frame with default settings
fromDataFrame(penguins, "penguins")
# read array as data.frame and without (default, added) row index
arr <- tiledb_array("penguins", as.data.frame=TRUE, extended=FALSE)</pre>
show(arr)
                                             # some array information
```

Introductory Example (cont.)

```
> df <- arr[]
> str(df)
'data.frame': 344 obs. of 8 variables:
$ species
                 : chr "Adelie" "Adelie" "Adelie" "Adelie" ...
$ island
                : chr "Torgersen" "Torgersen" "Torgersen" "Torgersen" ...
$ bill length mm : num 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
$ bill depth mm : num 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
$ flipper length mm: int 181 186 195 NA 193 190 181 195 193 190 ...
$ body mass g : int 3750 3800 3250 NA 3450 3650 3625 4675 3475 4250 ...
$ sex
                : chr "male" "female" "female" NA ...
$ year
```

Introductory Example (cont.)

Key Features

- We will discuss available options to create arrays
 - dense arrays versus sparse arrays
 - one or multiple indices (on sparse arrays)
 - options for creating and accessing arrays
 - but we mention tuning (tile extent, tile layout, ...) only in passing
- We will show different ways to read arrays back into R

Dense Arrays

Dense Data

The introductory example quickstart_dense.R creates an array with two integer domains and a single integer attribute:

```
# The array will be 4x4 with dims "rows" and "cols" and domain [1,4]
dom <- tiledb domain(dims = c(tiledb dim("rows", c(1L, 4L), 4L, "INT32"),</pre>
                               tiledb dim("cols", c(1L, 4L), 4L, "INT32")))
# The array will be dense with a single attribute "a" so
# each cell (i,j) cell can store an integer.
schema <- tiledb array schema(dom, attrs=c(tiledb attr("a", type="INT32")))</pre>
# Create the (empty) array on disk.
uri <- "quickstart dense"</pre>
tiledb arrav create(uri. schema)
```

Having created the array we can now open it for writing and add data.

```
# equivalent to matrix(1:16, 4, 4, bvrow=TRUE)
data \leftarrow array(c(c(1L, 5L, 9L, 13L),
                c(2L, 6L, 10L, 14L).
                c(3L. 7L. 11L. 15L).
                c(4L. 8L. 12L. 16L)), dim = c(4.4))
# Open the array and write to it.
A <- tiledb array(uri = uri)
A[] <- data
```

Data can be read back with different convenience wrappers:

```
arr <- tiledb_array(uri); arr[]  # list of columns

arr <- tiledb_array(uri, as.data.frame=TRUE); arr[] # a data.frame

arr <- tiledb_array(uri, as.matrix=TRUE); arr[] # a matrix

arr <- tiledb_array(uri, as.array=TRUE); arr[] # an array</pre>
```



[tile]DB

A data.frame example for dense arrays:

```
library(tiledb)
                         # load our package
uri <- tempfile()
                         # any local directory, more later on cloud access
## any data.frame, data.table, tibble ...; here we use penguins raw
fromDataFrame(palmerpenguins::penguins raw. uri)
# we want a data.frame, and we skip the implicit row numbers added as index
x <- tiledb array(uri, as.data.frame = TRUE, extended = FALSE)
newdf \leftarrow x[]
                         # full array (we can index rows and/or cols too)
```

```
> str(newdf[, 1:14]) # omitting last three cols for brevity
'data.frame': 344 obs. of 17 variables:
                    : chr "PAL0708" "PAL0708" "PAL0708" "PAL0708" ...
$ studvName
$ Sample Number
                   : num 1 2 3 4 5 6 7 8 9 10 ...
$ Species
                    : chr "Adelie Penguin (Pygoscelis adeliae)" ...
$ Region
                    : chr "Anvers" "Anvers" "Anvers" "Anvers" ...
$ Island
                    : chr "Torgersen" "Torgersen" "Torgersen" ...
$ Stage
                    : chr "Adult, 1 Egg Stage" "Adult, 1 Egg Stage" ...
$ Individual ID
                    : chr "N1A1" "N1A2" "N2A1" "N2A2" ...
$ Clutch Completion : chr "Yes" "Yes" "Yes" "Yes" ...
$ Date Egg
                    : Date. format: "2007-11-11" ...
 $ Culmen Length (mm) : num 39.1 39.5 40.3 NA 36.7 39.3 38.9 39.2 34.1 42 ...
 $ Culmen Depth (mm) : num 18.7 17.4 18 NA 19.3 20.6 17.8 19.6 18.1 20.2 ...
$ Flipper Length (mm): num 181 186 195 NA 193 190 181 195 193 190 ...
$ Body Mass (g) : num 3750 3800 3250 NA 3450 ...
 $ Sex
                   : chr "MALE" "FEMALE" "FEMALE" NA ...
```



Sparse Arrays

Sparse Array: Numeric

```
library(tiledb)
                                        # TileDB package
library(Matrix)
                                        # for sparse matrix functionality
uri <- tempfile()</pre>
                                        # array location
set.seed(123)
                                        # fix RNG seed
mat <- matrix(0, nrow=8, ncol=20)</pre>
mat[sample(seg len(8*20), 15)] \leftarrow seg(1, 15)
spmat <- as(mat, "dgTMatrix") # new sparse 'dgTMatrix'</pre>
fromSparseMatrix(spmat, uri)
                                       # store the sparse matrix in TileDB
chk <- toSparseMatrix(uri)</pre>
                                       # and retrieve it to check
```

Sparse Array: Numeric (cont.)

```
> chk # to check retrieved sparse matrix
 8 x 20 sparse Matrix of class "dgTMatrix"
 [3.] . . . . . 5 . . . . . 10 14 . . . . . . .
 [6,] . 2 . . . . . . . . . . . 4 . . . . .
 [8,] . . . . . . . . . . . . . . . . 6
[tile]DB
```

Sparse Array: Data Frame

```
library(tiledb)
                     # load our package
## now sparse with a character and integer ('year') index colum
## with wider range than seen in the data for year we allow appending
fromDataFrame(palmerpenguins::penguins, uri, sparse = TRUE,
            col index = c("species". "vear").
            tile domain=list(vear=c(2000L. 2021L)))
x <- tiledb array(uri, as.data.frame = TRUE, extended = FALSE)
newdf \leftarrow x[]
                     # full array (we can index rows and/or cols too)
```

Sparse Array: Data Frame (cont.)

Now we retrieve with two constraints: 'years' from 2007 to 2008 (both included), and 'species' equal to "Gentoo" (given as lower and upper range which implies equality). Note that both are *dimension* columns.

Sparse Array: Data Frame (cont.)

```
qc <- tiledb_query_condition_init("body_mass_g", 6000, "INT32", "GE")
query_condition(x) <- qc
newdf <- x[]</pre>
```

This selects rows based on the given attribute value, here body_mass_g which is required to be greater or equal to 6000 (grams).

Query conditions on attributes can also be combined (via standard Boolean operators).

```
Also (but not on CRAN yet): qc <- parse_query_condition(body_mass_g >= 6000)
```

Sparse Array: Select Attribute Columns

```
x <- tiledb_array(uri, as.data.frame = TRUE, extended = FALSE)
attrs(x) <- c("island", "sex")</pre>
```

This results in just the two selected attribute columns being returned (along with the two dimension columns).

Column selections can be combined with row selections.

Sparse Array: Incremental Writes

Setting the initial *domain* of the dimension columns (to ranges that accomodate future writes) allows incremental writes in batches.

As TileDB is serverless and inherently parallel, multiple writes can be made at the same time.

fromDataFrame & tiledb_array

fromDataFrame

High-level Array Writer

- Helper function to create arrays from existing data.frame data in R
- Can write dense arrays as well as sparse arrays
 - can add ad-hoc row-indices (dense and sparse)
 - or can use multiple index colums (sparse)
 - these can use int, numeric, or char data
- Defaults to using a ZStd compression filter
- Can set different TileDB array attributes and parameters
- Can support append mode via explicit dimension domain values
- We will see some examples later

tiledb_array

High-level Array Reader

- General array accessor for both dense and sparse arrays
- Supports multiple options to return as
 - · data.frame
 - matrix
 - array
- Supports selection of row ranges (via dimension constraint)
- Supports selection of returned columns



Full TileDB API

Full API

```
dims <- c(tiledb dim("rows", c(1L, 4L), 4L, "INT32"),</pre>
           tiledb dim("cols", c(1L, 4L), 4L, "INT32"))
attrs <- tiledb attr("a", type = "INT32")</pre>
schema <- tiledb_array_schema(tiledb_domain(dims), attrs)</pre>
tiledb array create(uri. schema)
data <- 1:16
arr <- tiledb array(uri = uri)</pre>
gry <- tiledb query(arr, "WRITE")</pre>
arv <- tiledb guerv set lavout(grv. "ROW MAJOR")</pre>
gry <- tiledb query set buffer(gry, "a", data)</pre>
grv <- tiledb guerv submit(grv)</pre>
gry <- tiledb query finalize(gry)</pre>
stopifnot(tiledb query status(gry)=="COMPLETE")
```

This example shows "quickstart_dense"

Each key function in the underlying TileDB Embedded (C++) API has been wrapped and is accessible directly.

This is useful when the higher-level functions need to be tweaked or customized.



Full API (using R 4.1.0 pipe)

```
dims <- c(tiledb dim("rows", c(1L, 4L), 4L, "INT32"),</pre>
          tiledb dim("cols", c(1L, 4L), 4L, "INT32"))
attrs <- tiledb attr("a", type = "INT32")</pre>
schema <- tiledb_array_schema(tiledb_domain(dims), attrs)</pre>
tiledb array create(uri, schema)
data <- 1:16
tiledb arrav(uri = uri) |>
    tiledb query("WRITE") |>
    tiledb guery set layout("ROW MAJOR") |>
    tiledb query set buffer("a", data) |>
    tiledb querv submit() |>
    tiledb query finalize()
stopifnot(tiledb query status(gry)=="COMPLETE")
```

This example shows "quickstart_dense" with the native pipe.

As many of the TileDB APi functions operate on the query type argument and return it, this style is easily supported.



Full API

Another example: retrieve the default configuration, overriden number of threads and asking for fragment meta-data consolitation (useful after many chunks have been written):

```
cfg <- tiledb_config()
cfg["sm.num_reader_threads"] <- 8
cfg["sm.num_writer_threads"] <- 8
cfg["vfs.num_threads"] <- 8
cfg["sm.consolidation.mode"] <- "fragment_meta"
ctx <- tiledb_ctx(cfg)
array_consolidate(uri=uri, cfg=cfg)</pre>
```

S3

```
uri <- "s3://namespace/bucket"</pre>
                                        # change URI as needed
## you need either these two environment variables
##
    AWS SECRET ACCESS KEY
##
    AWS ACCESS KEY ID
## or set this in the TileDB config object
fromSparseMatrix(spmat. uri) # stored
chk <- toSparseMatrix(uri) # retrieved</pre>
## lazy eval: e.g. for subsets only requested data transferred to client
```

[tile]DB

S3 (cont.)

```
> pp <- tiledb array("s3://tiledb-conferences/useR-2021/palmer penguins", as.data.frame=TRUE)
> dat <- pp[]
> head(dat)
  species year
                 island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
                                                                                      sex
1 Adelie 2007
                 Dream
                                  36.0
                                                17.9
                                                                   190
                                                                              3450 female
  Adelie 2007
                  Dream
                                  42.3
                                                21.2
                                                                   191
                                                                              4150
                                                                                     male
  Adelie 2007 Torgersen
                                  40.3
                                                18.0
                                                                   195
                                                                              3250 female
  Adelie 2007 Torgersen
                                  34.6
                                                21.1
                                                                   198
                                                                                     male
                                                                              4400
  Adelie 2007 Torgersen
                                  36.6
                                                17.8
                                                                   185
                                                                              3700 female
  Adelie 2007 Torgersen
                                  36.7
                                                                              3450 female
                                                19.3
                                                                   193
>
```

Arrow

Arrow

```
suppressMessages( { library(tiledb); library(arrow) } )
val <- 1:3  # arbitrary, could be rnorm() too</pre>
tvp <- int8() # any Arrow type</pre>
vec <- Array$create(val. tvp) # Arrow vector</pre>
aa <- tiledb arrow arrav ptr()</pre>
as <- tiledb arrow schema ptr()
on.exit( { tiledb arrow array del(aa); tiledb arrow schema del(as) } )
arrow:::ExportArray(vec, aa, as) # export Arrow to TileDB
newvec <- arrow::Arrav$create(arrow:::ImportArrav(aa. as))</pre>
stopifnot(all.equal(vec, newvec))
print(newvec) # show round-turn
```

Arrow (cont.)

```
> print(newvec) # show round-turn
Array
<int8>
  1,
 2,
```

Additional examples demonstrate zero-copy transfer from Arrow into TileDB Arrays, and the inverse from TileDB to Arrow.

Additional higher-level functions will likely get added soon.



Time Travel

Time Travel

TileDB Arrays add content in immutable "layers" (or fragments).

We can access their content at points in time!



Time Travel (cont.)

```
D <- data.frame(key=1:10, value=1:10)
   uri <- tempfile()</pre>
   fromDataFrame(D, uri, col index="key",
                   sparse=TRUE. allows dups=FALSE)
   now <- Sys.time()</pre>
   Sys.sleep(60)
                                                # one minute
   arr <- tiledb arrav(uri)</pre>
   D$value <- 100 + D$value
   arr[] <- D
   then <- Sys.time()
[tile]DB
```

Time Travel (cont.)

```
## we have written twice
show(arr[])

arrEarlier <- tiledb_array(uri, timestamp=now)
show(arrEarlier[])

arrLater <- tiledb_array(uri, timestamp=then)
show(arrLater[])</pre>
```

Encryption

Encryption

TileDB Arrays support encryption. The underlying files are controlled by standard filesystem access control layers, and additionally the content can be encrypted using standard AES-256 technology.

Encryption (cont.)

[tile]DB

```
dom <- tiledb domain(dims = tiledb dim("rows", c(1L, 4L), 4L, "INT32"))</pre>
schema <- tiledb array schema(dom, attrs=tiledb attr("a", type = "INT32"),
                               sparse = TRUE)
uri <- tempfile()</pre>
enckey <- "0123456789abcdef0123456789ABCDEF"
invisible(tiledb array create(uri, schema, enckey)) # schema with key
arr <- tiledb array(uri, encryption key = enckey)
                                                        # open with key to
                                                        # write and read
arr[] <- data.frame(rows=1:4. a=101:104)
chk <- tiledb array(uri, encryption key = enckey, as.data.frame=TRUE)</pre>
chk[]
```

Applications

SQL

SQL

Setup

- TileDB integrates with different frontends as well as languages
- One example: MariaDB with TileDB accessed via a 'storage plugin'
- Due to architectural choices at MariaDB, plugins
 - have to be compiled with the exact configuration as the server itself
 - we need to consistently build MariaDB, TileDB plugin ... and TileDB
- One easy way to do this is via Docker container tiledb-mariadb-r
- See https://hub.docker.com/r/tiledb/tiledb-mariadb-r/



Setup (cont.)

We launch the container as a daemon, allow MariaDB to accept empty password, and name the running image 'tiledb-mariadb-r':

```
## line break for display here
docker run --name tiledb-mariadb-r -it -d --rm \
    -e MYSQL_ALLOW_EMPTY_PASSWORD=1 tiledb/tiledb-mariadb-r
```

If desired, we can mount local directories via the standard Docker option -v local:container to access host data in container.



We then start R via Docker connecting to this session:

```
docker exec -it -u root tiledb-mariadb-r R
and in R write
```

```
library(tiledb)
fromDataFrame(palmerpenguins::penguins_raw, "/tmp/penguinsraw")
```

to create a TileDB Array in the context of the container.

[tile]DB

We then start R again via the same command for another R shell but now access the data.

Note that per standard semantics this query did *not* yet materialize.

```
> library(RMariaDB)
> library(dplyr, warn.conflicts=FALSE)
> con <- DBI::dbConnect(RMariaDB::MariaDB(), dbname="test")</pre>
> tbl(con, "/tmp/penguinsraw") |> dplyr::select(contains("Length"))
# Source: lazv guerv [?? x 2]
# Database: mysql [@localhost:NA/test]
   `Culmen Length (mm)` `Flipper Length (mm)`
                   <fdh>>
                                          < [db] >
                    39.1
                                            181
                   39.5
                                            186
                   40.3
                                            195
                    NΔ
                                             NA
                   36.7
                                            193
                   39.3
                                            190
                    38.9
                                            181
 8
                   39.2
                                            195
                    34.1
                                            193
10
                    42
                                            190
# ... with more rows
>
```

By adding collect() to the pipeline we ensure an actual retrieval of the data.

```
> tbl(con, "/tmp/penguinsraw") |>
      dplyr::select(contains("Length")) |>
      collect()
# A tibble: 344 x 2
   `Culmen Length (mm)` `Flipper Length (mm)`
                  <dbl>
                                         <dbl>
                   39.1
                                           181
                   39.5
                                           186
                   40.3
                                           195
                   NA
                                            NA
                   36.7
                                           193
                   39.3
                                           190
                   38.9
                                           181
                   39.2
                                           195
 9
                   34.1
                                           193
10
                   42
                                           190
# ... with 334 more rows
```



Start with top right to launch container ad daemon.

Next bottom right to create an array.

Finally left pane to access it.



Data Science with Flights Data

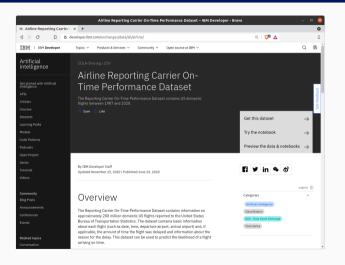
Data Science Example: Large Data Frame

We have already seen several examples for data.frames. The ability to index on different column types maps well with data.frame objects.

This example uses the well known flights data set.



Data Science Example: Large Data Frame



We use the full 'flights' data set (available under a permissible data license at the IBM site shown on the left.

It is available as both the full data set with 194 million rows, as well as in a 2 million row subset.



Creating the TileDB Array

The data comes as tar.gz containing a compressed csv file.

We cannot efficiently read all of the csv so we wrap a loop around, extracting chunks (via sed) which data.table::fread() can ingest. (We also skip a number of uninformative extra columns.)

We select four index columns. Three of these are character based and automatically obtain a <null,null> domain which we can append to.

For the fourth, we explicitly set an earlier start data and later (current) end date.

```
createIteratively <- function(csvxzfile, uri, n=100000, N=2000000) {</pre>
    stopifnot(`no csv.xz`=file.exists(csvxzfile))
   cmd <- paste0("xz -c -d ", csvxzfile, "| sed -n -e'1,", format(n+1, scientific=FALSE). "'p")</pre>
   cat(cmd, "\n")
   D <- fread(cmd=cmd, drop=c(48,57:109))
   cn <- colnames(D) # used below</pre>
   D <- filterData(D) # helper converting a few columns: utf8 char, bool to int, factor to char
   if (tiledb vfs is dir(uri)) tiledb vfs remove dir(uri)
   fromDataFrame(D, uri, sparse=TRUE,
                  col index = c("FlightDate", "Reporting Airline", "Origin", "Dest").
                  tile domain=list(FlightDate=c(as.Date("1970-01-01"), Sys.Date())))
   written <- n # keep track of data written
   ## remainder on next slide
```



```
## continued from previous slide
arr <- tiledb array(uri)</pre>
while (written < N) {</pre>
    cmd <- paste0("xz -c -d ", csvxzfile, "| sed -n -e'1d' -e'",</pre>
                   format(written+1+1, scientific=FALSE), ",",
                   format(min(written+n+1, N+1), scientific=FALSE), "'p")
    cat(cmd. "\n")
    D <- fread(cmd=cmd, drop=c(48,57:109))</pre>
    colnames(D) <- cn</pre>
                                      # assign colnames from first chunk
    D <- filterData(D)</pre>
    arr[] <- D
                                       # append chunk to TileDB array
    written <- written + n
invisible(NULL)
```



Operating on the full dataset–but selecting by dimensions 'FlightDate' and 'Reporting_Airline':



We we add additional conditions on attributes:

```
## as before
qc1 <- tiledb_query_condition_init("ArrDelay", 120, "FLOAT64", "GE")
qc2 <- tiledb_query_condition_init("DepDelay", 120, "FLOAT64", "GE")
query_condition(arr) <- tiledb_query_condition_combine(qc1, qc2, "AND")
res <- arr[]
print(dim(res)) ## now 21893 x 55</pre>
```



With not-yet-on-CRAN-but-at-GitHub current version can use a more direct approach:

```
qc <- parse_query_condition(ArrDelay >= 120.001 && DepDelay >= 120.001)
query_condition(arr) <- qc
res <- arr[]
print(dim(res)) ## now 21893 x 55</pre>
```

(The query condition parsing is independent of the array and does not know the underlying types which is why we used 120.001 to provide a hint that the delay columns are FLOAT64.)

Not that this is fully remote evaluation: we transmit the request including the selection constraints, and only the requested data is returned: here 22k rows out of 194 million.

Large Data Frame and SQL

We combine the two previous applications! Launching first in the directory above the 'flights' array:

```
docker run --name tiledb-mariadb-r -it -d --rm \
    -e MYSQL_ALLOW_EMPTY_PASSWORD=1 \
    -v $PWD:/mnt tiledb/tiledb-mariadb-r
```

to make the current ("outer") directory (accessed via shell variable $\protect\ PWD$) in the container a path $\protect\ mnt$. Then we launch R in the container via

```
docker exec -it -u root tiledb-mariadb-r R
```



Large Data Frame and SQL

[tile]DB

```
> library(RMariaDB); library(dplyr, warn.conflicts=FALSE)
> con <- DBI::dbConnect(RMariaDB::MariaDB(), dbname="test")</pre>
> tbl(con, "/mnt/airline") |> dplyr::select(contains("Dep"))
# Source: lazv guerv [?? x 7]
# Database: mysql [@localhost:NA/test]
   DepartureDelayGroups DepDel15 DepTime DepTimeBlk DepDelay DepDelayMinutes
                  <int>
                           <dbl> <int> <chr>
                                                       <dbl>
                                                                        <dbl>
                                    1402 1400-1459
                               0
                                    1750 1700-1759
                                    1108 1100-1159
                                     511 0001-0559
                                     928 0900-0959
                               Θ
                                    1631 1600-1659
             2147483647
                                    2111 2100-2159
 8
             2147483647
                               0
                                    1305 1300-1359
                                     858 0800-0859
                               0
10
                                     648 0600-0659
                               0
# ... with more rows, and 1 more variable: CRSDepTime <int>
>
```

LiDAR

LiDAR

- LiDAR stands for Light Detection and Ranging
- It is a method for determining ranges (often using lasers)
- Used in spatial analysis, forestry, or even autonomous driving
- Many (public) data sets via LAS or LAZ (compressed) files
- As these are multidimensional arrays use maps well to TileDB

Lidar Ingest

The PDAL (Point Data Abstraction Library) is central, and the pdal binary can be built with TileDB support.

We use a Docker container tiledb-geospatial to read LAS (or LAZ) files and create an array as described on the TileDB docs website.

Command:

```
pdal pipeline -i pipeline.json
```

where pdal may come from the tiledb-geospatial container, and the JSON control file shown to the right might control reading and writing steps.

```
{
    "type": "readers.las",
    "filename": "autzen.laz"
    },
    {
        "type": "writers.tiledb",
        "array_name": "autzen_tiledb",
        "chunk_size": 100000000
    }
}
```

Lidar

```
lasfile <- "LAS 17258975.las"
if (!file.exists(lasfile)) {
    ## note: the file is 451 mb
    op <- options()</pre>
                                        # store
    options(timeout=3600) # (much) more patience downloading
    lasfileurl <- file.path("https://clearinghouse.isgs.illinois.edu/las-east/cook/las/". lasfile)</pre>
    download.file(lasfileurl,lasfile)
    options(op)
                                        # reset
if (!dir.exists("las array")) {
    wd <- getwd()
    cmd <- paste0("docker run --rm -ti -u 1000:1000 -v ", wd, ":/data ",</pre>
                  "-w /data tiledb/tiledb-geospatial pdal pipeline -i pipeline.ison")
    system(cmd)
                                        # fancier return code check possible
```



LiDAR (cont.)

Note that we can loop similarly over many LAS or LAZ files, and can also inject them in parallel. The JSON file needs "append": true to append; this way we can store many LAS or LAZ files in a single TileDB Array, locally or in the cloud.

Being able to store many such files in a single (cloud-hosted or local) array shows one of the strengths of TileDB. And data requests will transfer only the requested subset.

LiDAR (cont)

We can then read from the LiDAR array. The following extracts just 100k rows of points from a well-known building:

```
library(tiledb)
arr <- tiledb_array("las_array", as.data.frame=TRUE)
selected_ranges(arr) <- list(X = cbind(1174100, 1174400), Y = cbind(1899125, 1899250))
L <- arr[]
## print(dim(L)) # 108655 x 15

library(lidR)
L$ScanAngleRank <- as.integer(L$ScanAngleRank)
LL <- LAS(L)
plot(LL) # open rgl device
## plot(LL, backend="lidRviewer") # if lidRviewer is installed</pre>
```

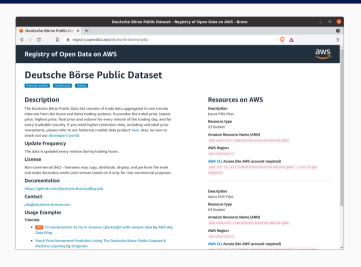


Finance / Time Series

Time Series

TileDB can also be used for financial data such as transactions data from an exchange, times and sales data from trades, or aggregates. In this example we will look at a data set provided (and regularly updated) by Deutsche Boerse covering one-minute bars of each stock and etf (for the stock exchanges) and each future (for the Eurex sister exchange focussing on derivatives).

Time Series



Provided by the exchange via AWS

"[...] provides the initial price, lowest price, highest price, final price and volume for every minute of the trading day, and for every tradeable security."

"If you need higher resolution data, including untraded price movements, please refer to our historical market data product here."



Time Series

List the files (here a small demo sample).

Helper function to construct datetime column, and remove date and time columns.

Simple injection loop. First file creates the array and defines the schema. We set minimum amd maximum time values.

Injection could run in parallel, or an automated script appending new data.

```
uri <- "dhoerse"
files <- list.files(pattern="2020-.*\\.csv") # files retrieved Fall of 2020
readAndAddDatetime <- function(file) { # simple helper</pre>
    D <- fread(file)
    setDT(D)
    D[, Datetime := as.POSIXct(paste(Date, Time))]
    D[, `:=`(Date = NULL, Time = NULL)]
    invisible(D)
n <- length(files)</pre>
for (i in seg len(n)) {
    D <- readAndAddDatetime(files[i])</pre>
    if (i == 1) {
        fromDataFrame(D. uri. sparse = TRUE.
                       col index=c("Mnemonic"."Datetime").
                       tile domain=list(Datetime=c(as.POSIXct("1970-01-01 00:00:00"). Sys.time())))
      else {
        arr <- tiledb_array(uri, as.data.frame = TRUE)</pre>
        arr[] <- D
        tiledb array close(arr)
```

Time Series

Simple usage example: one hour of BMW trades in one-minute bars



Time Series

```
suppressMessages({
    library(rtsplot)
                                             # for nicer financial plot
    library(xts)
                                            # used by rtsplot
setDT(BMW)
symbol <- "BMW"
rt <- as.xts(BMW[Mnemonic==symbol.
                 .(Datetime, Open=StartPrice, High=MaxPrice,
                   Low=MinPrice, Close=EndPrice, Volume=TradedVolume)])
cols <- rtsplot.colors(2)</pre>
lavout(c(1.1.1.1.2))
rtsplot(rt, type="n")
rtsplot.ohlc(rt, col=rtsplot.candle.col(rt))
rtsplot.legend(symbol, cols[1], list(rt))
rt <- rtsplot.scale.volume(rt)
rtsplot(rt, type = 'volume', plotX = FALSE, col = 'darkgray')
rtsplot.legend('Volume', 'darkgray', quantmod::Vo(rt))
```





GWAS

What is a GWAS?

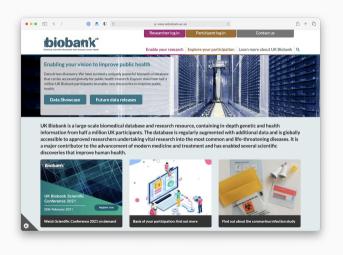
Overview

- GWAS: Genome-wide Association Study
- Used to identify regions of the genome that are associated with a particular trait (e.g., hair color)
- Requires:
 - sequencing data on a large population of samples to identify genetic variants
 - 2) measurements for the trait of interest across the same samples

GWAS Results Example

variant	beta	se	tstat	pval
1:15791:C:T	-1.70174e+01	5.66755e+01	-3.00260e-01	7.63979e-01
1:69487:G:A	-5.70053e-02	1.11014e-01	-5.13496e-01	6.07605e-01
1:69569:T:C	-2.30684e-03	1.99098e-02	-1.15865e-01	9.07760e-01
1:139853:C:T	-5.62416e-02	1.11017e-01	-5.06603e-01	6.12434e-01
1:692794:CA:C	7.72562e-04	9.22074e-04	8.37852e-01	4.02114e-01
1:693731:A:G	1.31202e-03	8.71218e-04	1.50596e+00	1.32078e-01
1:707522:G:C	8.77269e-04	9.79498e-04	8.95631e-01	3.70450e-01
1:717587:G:A	-8.32431e-05	2.33724e-03	-3.56160e-02	9.71589e-01
1:723329:A:T	-1.15975e-02	6.88597e-03	-1.68422e+00	9.21406e-02
1:730087:T:C	4.23934e-05	1.21371e-03	3.49286e-02	9.72137e-01

Data source: UK Biobank



About

Provides an incredibly rich source of biomedical data collected from hundred of thousands of volunteers in the United Kingdom.



UK Biobank GWAS Dataset Stats

- Contains ~12,000 GWAS results files
- Analyzed over >4,000 traits across >350,000 individuals
- Also includes different versions of each analysis (e.g., sex-specific results)
- Each file:
 - contains ~10 million rows
 - ~500Mb gzipped (1.7Gb uncompressed)

UK Biobank announcement:

http://www.nealelab.is/uk-biobank/ukbround2announcement

Data Accessibility Goals

- Available on a remote cloud bucket
- Facilitate comparisons across phenotypes
- Query variants by their genomic location
- Query traits by their descriptive names



Tutorial Files

Copy GWAS tutorial to your working directory

```
# library(tiledb.user2021)
file.copy(
  from = system.file("examples/exGWAS.R", package = "tiledb.user2021")
  to = "exGWAS.R"
)
```

Download GWAS results files

```
dir.create("gwas-tutorial/data", showWarnings = FALSE, recursive = TRU
download_gwas_files("gwas-tutorial/data")
```

Extracting Genomic Location Data

variant	becomes	chr	pos	ref	alt
1:15791:C:T		1	15791	С	Т
1:69487:G:A		1	69487	G	Α
1:69569:T:C		1	69569	Т	C
1:139853:C:T		1	139853	C	Т
1:692794:CA:C		1	692794	CA	C
1:693731:A:G		1	693731	Α	G
1:707522:G:C		1	707522	G	C
1:717587:G:A		1	717587	G	Α



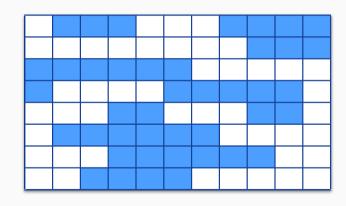
Array Dimensions

- 1. GWAS Phenotype (e.g., Ventricular rate)
- 2. Variant chromosome (e.g., chromosome 1)
- 3. Chromosome location (e.g., 43,113,410 bp)

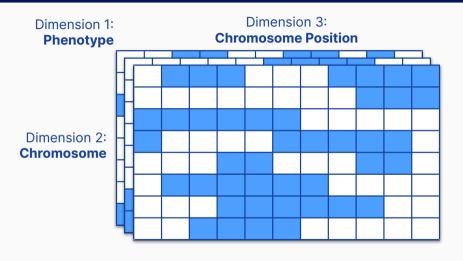
Array Layout

Dimension 3: **Chromosome Position**

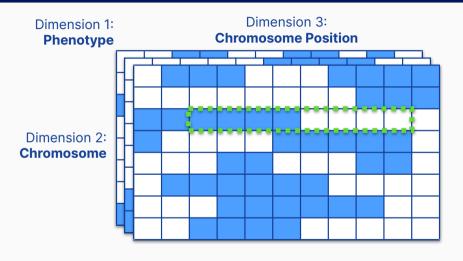
Dimension 2: **Chromosome**



GWAS Array Layout



GWAS Array Layout



GWAS Array Dimension 1

Phenotype (the descriptive name for each analyzed trait)

```
dim_pheno <- tiledb_dim(
  name = "phenotype",
  domain = NULL,
  tile = NULL,
  type = "ASCII"
)</pre>
```



GWAS Array Dimension 2

Chromosome labels

```
dim_chr <- tiledb_dim(
    name = "chr",
    domain = NULL,
    tile = NULL,
    type = "ASCII"
)</pre>
```



GWAS Array Dimension 3

Chromosome position

```
dim_pos <- tiledb_dim(
  name = "pos",
  domain = c(1L, 249250621L),
  tile = 1e5L,
  type = "UINT32"
)</pre>
```

GWAS Array Attributes

```
attr_filters <- tiledb_filter_list(tiledb_filter("ZSTD"))

all_attrs <- list(
    ref = tiledb_attr("ref", type = "CHAR", filter_list = attr_filters),
    alt = tiledb_attr("alt", type = "CHAR", filter_list = attr_filters),
    minor_AF = tiledb_attr("minor_AF", type = "FLOAT64", filter_list = attr_filters),
    pval = tiledb_attr("pval", type = "FLOAT64", filter_list = attr_filters),
    tstat = tiledb_attr("tstat", type = "FLOAT64", filter_list = attr_filters),
    se = tiledb_attr("se", type = "FLOAT64", filter_list = attr_filters),
    beta = tiledb_attr("beta", type = "FLOAT64", filter_list = attr_filters)
)</pre>
```



GWAS Array Creation

[tile]DB

```
# assemble the schema
gwas schema <- tiledb arrav schema(
  domain = tiledb domain(dims = c(dim pheno, dim chr, dim pos)),
  attrs = all attrs.
  sparse = TRUE,
  allows dups = TRUE
# create the array
gwasdb uri <- "ukbiobank-gwasdb"</pre>
tiledb array create(gwasdb uri, schema = gwas schema)
```

Download GWAS Results Files

A set of 6 slimmed down results files created for this tutorial can be downloaded using the helper package:

```
dir.create("gwas_files", showWarnings = FALSE)
tiledb.user2021::download_gwas_files("gwas_files")
```

These files were processed to:

- include only chromosomes 20, 21, and 22
- remove variants flagged as low confidence
- parse chromosomal location from the variant identifiers
- add a phenotype column with GWAS trait description



Open GWAS Array

Let's return the results as a data.frame that includes the subset of attributes we're interested in.

```
gwasdb <- tiledb_array(
   gwasdb_uri,
   is.sparse = TRUE,
   as.data.frame = TRUE,
   attrs = c("beta", "se", "tstat", "pval")
)</pre>
```



GWAS Query #1

Use [] indexing to query the first 2 dimensions (e.g., phenotype and chr).

```
gwasdb["Water intake", "20"]
```

```
# A tibble: 295,761 x 7
  phenotype
            chr
                    pos
                            beta se tstat pval
       <chr> <int>
                            <dbl> <dbl>
                                         <dbl> <dbl>
  <chr>
1 Water intake 20
                                 0.00246 0.812 0.417
                   61098 0.00199
2 Water intake 20
                  61270 -0.00113
                                 0.00719 -0.157 0.876
3 Water intake 20
                  61795 0.000381
                                 0.00218 0.175 0.861
4 Water intake 20
                  62731 -0.00200
                                 0.00328 -0.611 0.541
5 Water intake 20
                   63231 0.00219
                                 0.00683 0.320 0.749
```



GWAS Query #2

Use selected_ranges to query all 3-dimensions and extract data for a specific genomic region.

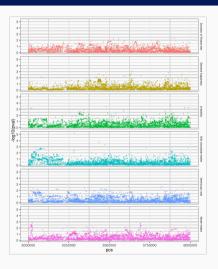
```
selected_ranges(gwasdb) <- list(
  phenotype = cbind("Water intake", "Water intake"),
  chr = cbind("20", "20"),
  pos = cbind(5e6, 6e6)
)
gwasdb[]</pre>
```

```
# A tibble: 5.198 x 7
  phenotype
              chr
                       pos
                           beta
                                        se tstat
                                                   pval
  <chr>
        <chr>
                     <int>
                             <dbl> <dbl>
                                           <dbl> <dbl>
1 Water intake 20
                   5000142 0.0138 0.0103
                                           1.34 0.180
2 Water intake 20
                   5000146 -0.00457 0.00529 -0.864 0.388
3 Water intake 20
                   5000279 0.00523 0.0181
                                           0.288 0.773
4 Water intake 20
                   5000280 -0.00605 0.00246 -2.46 0.0139
5 Water intake 20
                   5000337 -0.00459 0.00529 -0.867 0.386
```

GWAS Query #3

Examine p-values across all phenotypes for the same genomic region.

```
selected_ranges(gwasdb) <- list(
   phenotype = NULL,
   chr = cbind("20", "20"),
   pos = cbind(5e6, 6e6)
)
gwas_results <- gwasdb[]
manhattan_plot(gwas_results)</pre>
```



GWAS Resources

- 1. UK Biobank (https://www.ukbiobank.ac.uk)
- 2. Neale Lab UK Biobank GWAS results (https://www.nealelab.is/uk-biobank)
- 3. GWAS Results Manifest



Wrap-Up

In Summary

TileDB

- an open-source embeddable storage engine
- an open-source format for modeling any type of data
- fully cloud-native on AWS, GCS, Azure
- · limitless scalability
- offers time travel
- offers Encryption



In Summary

TileDB R Package

- available on CRAN, and already used by Bioconductor
- high-level R-friendly interface for creating/query TileDB arrays
- also includes low-level access to the full TileDB API
- fully interoperable with DBI, Arrow, ...



In Summary

Use cases

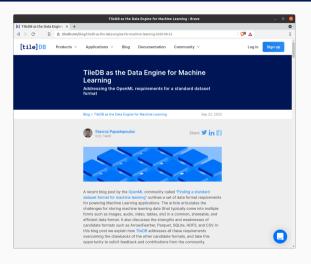
Use cases covered today

- Data Frames
- LiDAR and Geospatial uses
- Finance and Time Series
- Population Genomics and GWAS



Further Resources

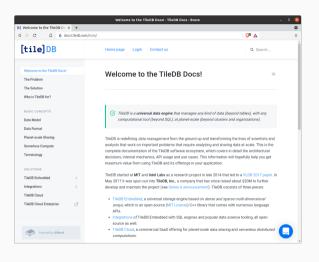
Resources



Blog post describing how TileDB answers the data format requirements for scientific data as layed out in an earlier post by the OpenML team.



Documentation



Extensive documentation on TileDB, APIs, Usage, and more

```
docs.tiledb.com
github.com/TileDB-Inc/TileDB-R
github.com/TileDB-Inc/TileDB
```



Talk to TileDB

```
email helloatiledb.com
  web https://tiledb.com/
 docs https://docs.tiledb.com/main
forum https://forum.tiledb.com/
qithub https://github.com/TileDB-Inc/TileDB
twitter https://twitter.com/tiledb
 slack https://tiledb-community.slack.com/
  jobs https://apply.workable.com/tiledb/
```

we're hiring!!