# 7. Working with Big Data

**Thomas Lumley**
**Ken Rice**

Universities of Washington and Auckland

*Seattle, July 2015*

# Large data

Really?

*R is well known to be unable to handle large data sets*

Solutions:

- Get a bigger computer: 8-core Linux computer with 32Gb memory for $<$ \$3000
- Don't load all the data at once, i.e. use methods from the mainframe days

# Large data

Data won't fit in memory on current computers: R can comfortably handle data up to

- About 1/3 of physical RAM
- About 10% of address space (ie, no more than 400MB for 32-bit R, no real constraint for 64-bit R)

R can't (currently) handle a single matrix with more than $2^{31} - 1 \approx 2$ billion entries *even if* your computer has memory for it.

Storing data on disk means extra programming work, but has the benefit of making you aware of data reads/writes in algorithm design.

# Storage formats

R has two convenient data formats for large data sets

- For ordinary large data sets, direct interfaces to relational databases allow the problem to be delegated to the experts.

- For very large 'array-structured' data sets the `ncdf` package provides storage using the netCDF data format.

# SQL-based interfaces

Relational databases are the natural habitat of large datasets.

- Optimized for loading subsets of data from disk
- Fast at merging, selecting
- Standardized language (SQL) and protocols (ODBC, JDBC)

# Elementary SQL

Basic statement: `SELECT var1, var2 FROM table`

- `WHERE condition` to choose rows where condition is true
- `table1 INNER JOIN table2 USING(id)` to merge two tables on a identifier
- Nesting: `table` can be a complete `SELECT` statement

# R database interfaces

- RODBC package for ODBC connections (mostly Windows)

- DBI: standardized wrapper classes for other interfaces
  - RSQLite: small, zero-configuration database for embedded storage
  - RJDBC: Java interface
  - also for Oracle, MySQL, PostgreSQL.

# Setup: DBI

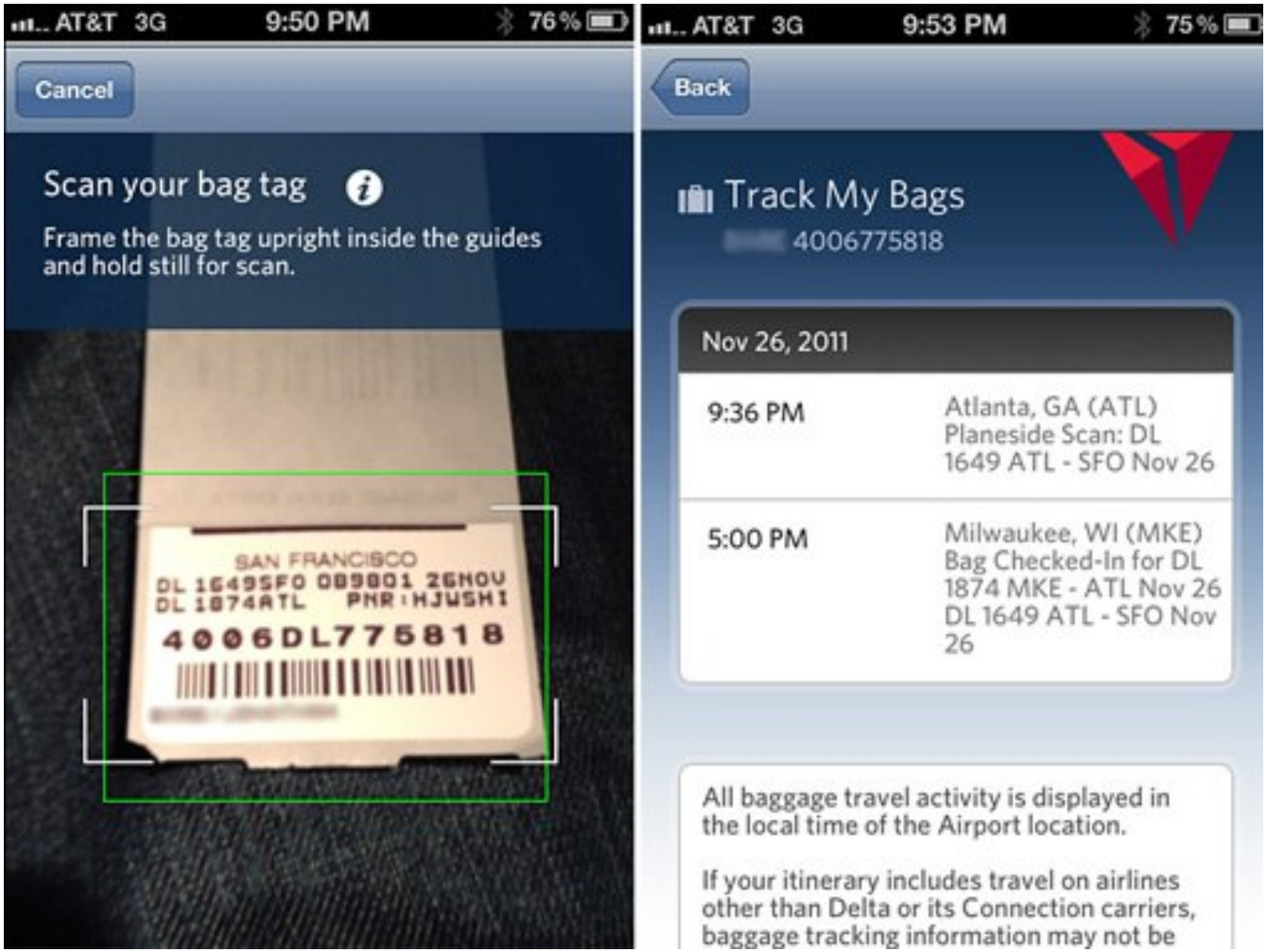
Needs DBI package + specific interface package

```
library("RSQLite") ## also loads DBI package

sqlite <- dbDriver("SQLite")
conn <- dbConnect(sqlite, "example.db")

dbListTables(conn) ## what tables are available?
                   ## see also dbListFields()

dbDisconnect(conn) ## when you are done
```

Now use `conn` to identify this database connection in future requests

# Setup: DBI

How to think about `conn`, and what it does;



# Queries: DBI

- `dbGetQuery(conn, "select var1, var2, var22 from smalltable")`: runs the SQL query and returns the results (if any)

- `dbSendQuery(conn, "select var1, var2, var22 from hugetable")` runs the SQL and returns a result set object

- `fetch(resultset, n=1000)` asks the database for the next 1000 records from the result set

- `dbClearResult(resultset)` releases the result set

# Whole tables: DBI

- `dbWriteTable(conn, "tablename", dataframe)` writes the whole data frame to a new database table (use `append=TRUE` to append to existing table)

- `dbReadTable(conn, "tablename")` reads a whole table

- `dbDropTable(conn, "tablename")` deletes a table.

# Setup: ODBC

This just needs the RODBC package. The Database must be given a "Data Source Name" (DSN) using the ODBC adminstrator on the Control Panel.

```
library("RODBC")
```

```
conn <- odbcConnect(dsn)
```

```
close(conn) ## when you are done
```

Now use conn to identify this database connection in future requests

# Queries: ODBC

- `sqlQuery(conn, "select var1, var2, var22 from smalltable")`: runs the SQL query and returns the results (if any)

- `odbcQuery(conn, "select var1, var2, var22 from hugetable")` runs the SQL and returns a result set object

- `sqlGetResults(con, max=1000)` asks the database for the next 1000 records from the result set

# Whole tables: ODBC

- `sqlSave(conn, dataframe, "tablename")` writes the whole data frame to a new database table (use `append=TRUE` to append to existing table)

- `sqlFetch(conn, "tablename")` reads a whole table

# Example: SQLite and Bioconductor

Bioconductor's AnnotationDBI system maps from one system of identifiers (eg probe ID) to another (eg GO categories).

Each annotation package contains a set of two-column SQLite tables describing one mapping.

'Chains' of tables allow mappings to be composed so, e.g. only gene IDs need to be mapped directly to GO categories.

The original annotation system kept all tables in memory; they are now too large for this.

# AnnotationDBI

```
> library("hgu95av") # a Bioconductor package: see Session 8
> hgu95av2CHR[["1001_at"]]
[1] "1"
> hgu95av2OMIM[["1001_at"]]
[1] "600222"
> hgu95av2SYMBOL[["1001_at"]]
[1] "TIE1"
> length(hgu95av2GO[["1001_at"]])
[1] 16
```

# Under the hood

```
> ls("package:hgu95av2.db")
 [1] "hgu95av2"               "hgu95av2_dbconn"       "hgu95av2_dbfile"
 [4] "hgu95av2_dbInfo"        "hgu95av2_dbschema"     "hgu95av2ACCNUM"
 [7] "hgu95av2ALIAS2PROBE"    "hgu95av2CHR"           "hgu95av2CHRLENGTHS"
[10] "hgu95av2CHRLOC"         "hgu95av2CHRLOCEND"     "hgu95av2ENSEMBL"
> hgu95av2_dbconn()
<SQLiteConnection: DBI CON (7458, 1)>
> dbGetQuery(hgu95av2_dbconn(), "select * from probes limit 5")
    probe_id gene_id is_multiple
1    1000_at    5595           0
2    1001_at    7075           0
3  1002_f_at    1557           0
4  1003_s_at     643           0
5    1004_at     643           0
```

# Under the hood

The `[[` method calls the `mget` method, which also handles multiple queries.

These (eventually) produce SQLite `SELECT` statements with `INNER JOIN`s across the tables needed for the mapping.

# netCDF


netCDF was designed by the NSF-funded UCAR consortium, who also manage the National Center for Atmospheric Research.

Atmospheric data are often array-oriented: eg temperature, humidity, wind speed on a regular grid of $(x, y, z, t)$.

Need to be able to select 'rectangles' of data – e.g. range of $(x, y, z)$ on a particular day $t$.

Because the data are on a regular grid, the software can work out where to look on disk without reading the whole file: efficient data access.

Many processes can read the same netCDF file at once: efficient parallel computing.

# Current uses in biology

- Whole-genome genetic data (us and people we talk to)
  - Two dimensions: genomic location $\times$ sample, for multiple variables
  - Data sizes in tens to thousands of gigabytes.
- Flow cytometry data (proposed new FCS standard)
  - 5–20 (to 100, soon) fluorescence channels $\times$ 10,000–10,000,000 cells $\times$ 5–5000 samples
  - Data sizes in gigabytes to thousands of gigabytes.

# Using netCDF data

Interact with netCDF files via the `ncdf` package, in much the same way as databases;

`open.ncdf()` opens a netCDF file and returns a connection to the file (rather than loading the data)

`get.var.ncdf()` retrieves all or part of a variable.

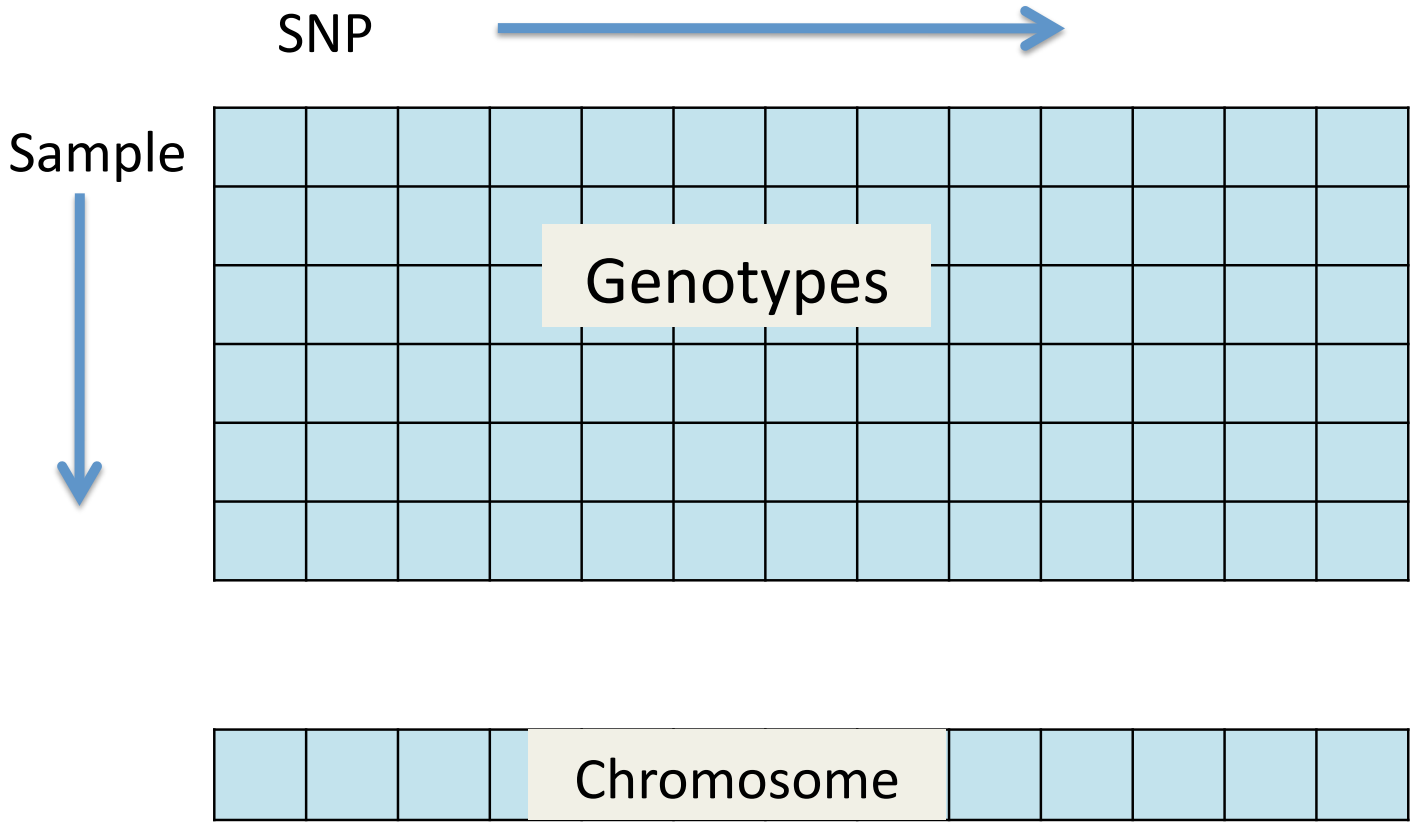
`close.ncdf()` closes the connection to the file.

To see what variables a file contains, and their dimensions, e.g.

```
conn <- open.ncdf("mynetCDFfile.nc")
conn
```

# Dimensions

Variables can use one or more array dimensions of a file



# Dimensions

To read data that has given dimensions, netCDF's `get.var.ncdf()` function uses an unusual (but logical) system, with these arguments;

- `nc` – the netCDF connector object, returned by `create.ncdf`
- `varid` – the name of the variable you want to read, e.g. `"genotype"` or `"chromosome"`
- `start` – a vector containing the location at which you start reading, e.g. `c(1,1)` or just `c(10)`
- `count` – *how far* you want to read, in each dimension, e.g. `c(50,100)` for a 50×100 rectangle, just `99` for 99 elements in a 1D object

For `count`, -1 is a special 'code', that means 'all the entries in that dimension until the end'. **It's very useful!**

# Example

Finding long homozygous runs (possible deletions)

```
library("ncdf")
nc <- open.ncdf("hapmap.nc")

## read all of chromosome variable
chromosome <- get.var.ncdf(nc, "chr", start=1, count=-1)
## set up list for results
runs<-vector("list", nsamples)

for(i in 1:nsamples}{
   ## read all genotypes for one person
   genotypes <- get.var.ncdf(nc, "geno", start=c(1,i),count=c(-1,1))
   ## zero for htzygous, chrm number for hmzygous
   hmzygous <- genotypes != 1
   hmzygous <- as.vector(hmzygous*chromosome)
```

# Example

```
    ## consecutive runs of same value
    r <- rle(hmzygous)
    begin <- cumsum(c(1, r$lengths))
    end   <- cumsum(r$lengths)
    long  <- which ( r$lengths > 250 & r$values !=0)
    runs[[i]] <- cbind(begin[long], end[long], r$lengths[long])
}

close.ncdf(nc)
```

Notes

- `chr` uses only the 'SNP' dimension, so `start` and `count` are single numbers
- `geno` uses both SNP and sample dimensions, so `start` and `count` have two entries
- `rle()` compresses runs of the same value to a single entry

# Advanced: creating netCDF files

Creating files is more complicated;

- Define dimensions
- Define variables and specify which dimensions they use
- Create an empty file
- Write data to the file.

# Advanced: defining dimensions

Specify the name of the dimension, the units, and the allowed values in the `dim.def.ncdf()` function.

One dimension can be 'unlimited', allowing expansion of the file in the future. An unlimited dimension is important, otherwise the maximum variable size is 2Gb.

```
snpdim    <- dim.def.ncdf("position","bases", positions)
sampledim <- dim.def.ncdf("seqnum","count",1:10, unlim=TRUE)
```

# Advanced: defining variables

Variables are defined by name, units, and dimensions

```
varChrm <- var.def.ncdf("chr","count",dim=snpdim,
         missval=-1, prec="byte")
varSNP <- var.def.ncdf("SNP","rs",dim=snpdim,
        missval=-1, prec="integer")
vargeno <- var.def.ncdf("geno","base",dim=list(snpdim, sampledim),
        missval=-1, prec="byte")
vartheta <- var.def.ncdf("theta","deg",dim=list(snpdim, sampledim),
        missval=-1, prec="double")
varr <- var.def.ncdf("r","copies",dim=list(snpdim, sampledim),
       missval=-1, prec="double")
```

# Advanced: creating the file

The file is created by specifying the file name ad a list of variables.

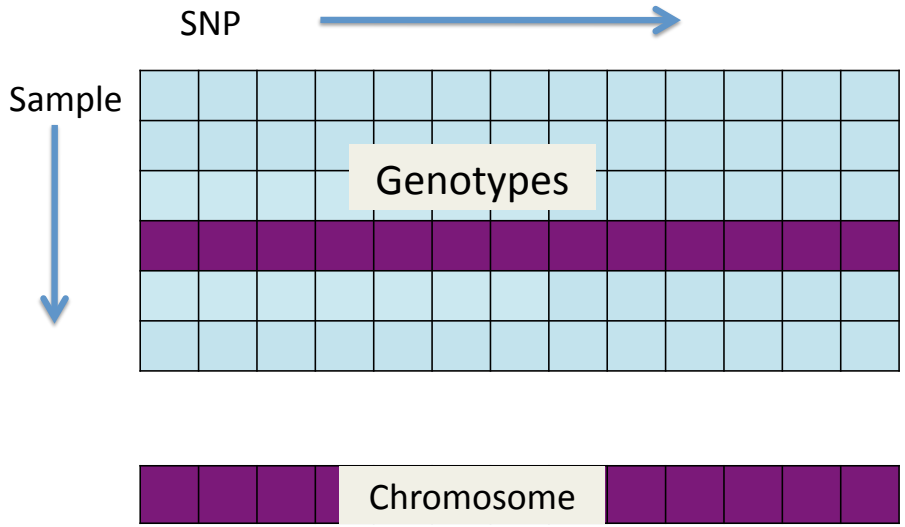
```
genofile<-create.ncdf("hapmap.nc", list(varChrm, varSNP, vargeno,
                    vartheta, varr))
```

The file is empty when it is created. Data can be written using `put.var.ncdf()`. Because the whole data set is too large to read, we might read raw data and save to netCDF for one person at a time;

```
for(i in 1:4000){
    temp.geno.data <- readRawData(i) ## somehow
    put.var.ncdf(genofile, "geno", temp.geno.data,
           start=c(1,i), count=c(-1,1))
}
```

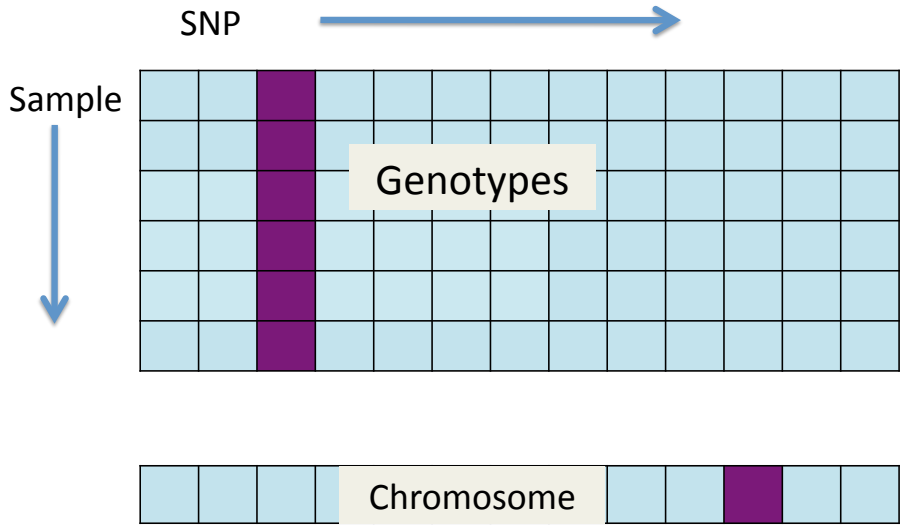
# Efficient use of netCDF

Read all SNPs, one sample;



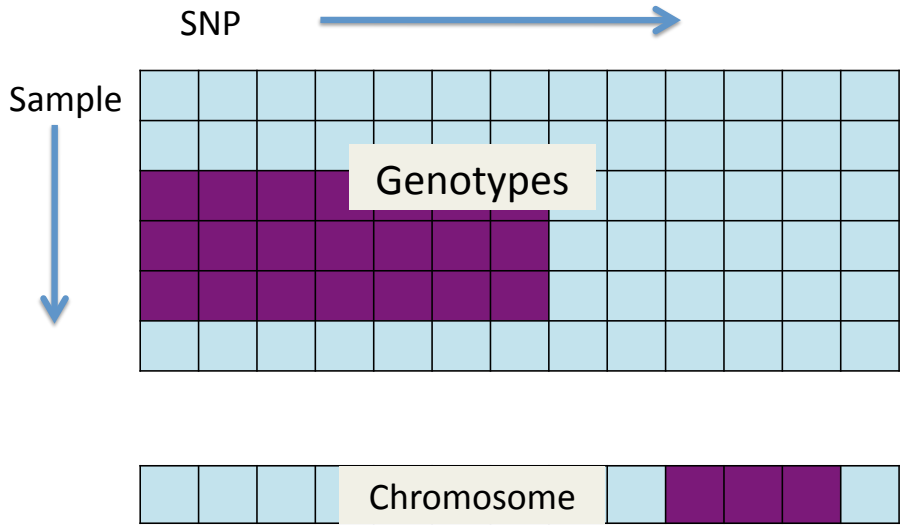
# Efficient use of netCDF

Read all samples, one SNP;



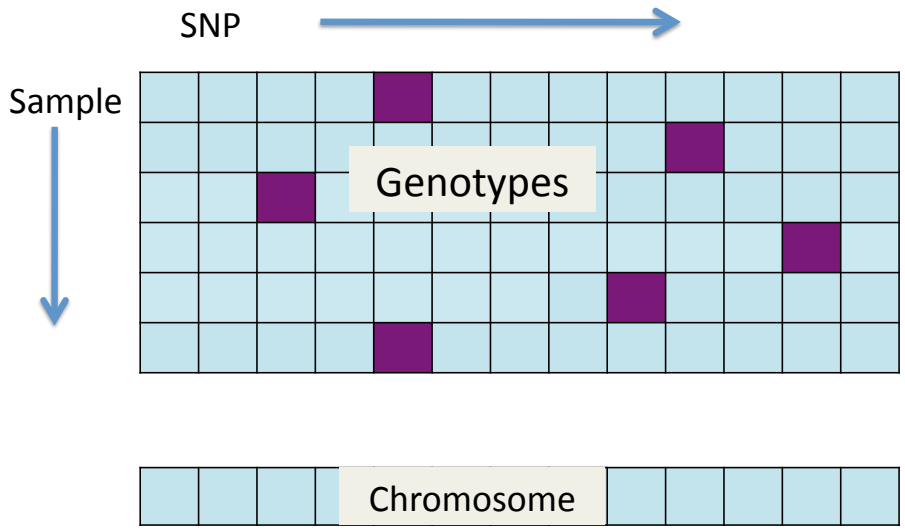
# Efficient use of netCDF

Read some samples, some SNPs;



# Efficient use of netCDF

Random access is not efficient: e.g. read probe intensities for all missing genotype calls;



# Efficient use of netCDF

- Association testing: read all data for one SNP at a time
- Computing linkage disequilibrium near a SNP: read all data for a contiguous range of SNPs
- QC for aneuploidy: read all data for one individual at a time (and parents or offspring if relevant)
- Population structure and relatedness: read all SNPs for two individuals at a time.