Best Practices on the DNAnexus Platform

Ted Laderas

1/23/23

Table of contents

Preface

This is a companion book to Bash for Bioinformatics.

In this book we attempt to highlight best practices on the DNAnexus platform given some previous knowledge.

Learning Objectives

After reading this book and doing the exercises, you should be able to:

- **Explain** the object based filesystem on the DNAnexus platform and how it differs from Linux/Unix POSIX filesystems.
- **Explain** key details of the UKB Research Analysis Platform and how these details impact your work.
- **Utilize** Projects and Organizations to effectively manage data for your group.
- **Use** Spark to connect, extract, and manipulate Apollo Datasets on the platform.
- **Utilize** Hail to load, query, model, annotate, and visualize large-scale genomics data such as the Exome Data on the UK Biobank Research Analysis Platform.
- **Build** native DNAnexus apps effectively that manage inputs, outputs, and work with batch mode.
- **Utilize** existing and **build** native DNAnexus workflows using Workflow Description Language (WDL).
- **Execute** NextFlow pipelines on the DNAnexus Platform.

Our goal is to bring information together in a task-oriented format to achieve things on the platform.

Four Levels of Using DNAnexus

One way to approach learning DNAnexus is to think about the skills you need to process a number of files. Ben Busby has noted there are 4 main skill levels in processing files on the DNAnexus platform:

Level	# of Files	Skill
1	1	Interactive Analysis (Cloud Workstation, JupyterLab)
2	1-50 Files	`dx run`, Swiss Army Knife
3	50-1000 Files	Building your own apps
4	1000+ Files, multiple steps	Using WDL (Workflow Description Language)

We'll be covering mostly level 3 and 4 in this book. But you will need to be at level 2 before you can tackle these topics.

The key is to gradually build on your skills.

Prerequisites

Before you tackle this book, you should be able to accomplish the following:

- Utilize Basic Bash Skills for DNAnexus
- Understand the basic architecture of Cloud Computing
- Know how to edit and utilize JSON Files
- Be familiar with `dx-toolkit` commands, including:
  - `dx run`
  - `dx find data` and `dx find jobs`/`dx watch`

We recommend reviewing Bash for Bioinformatics if you need to brush up on these prerequisite skills.

Want to be a Contributor?

This is the first draft of this book. It's not going to be perfect, and we need help.

If you find a problem/issue or have a question, you can file it as an issue using this link.

In your issue, please note the following:

- Your Name
- What your issue was
- Which section, and line you found problematic or wouldn't run

If you're quarto/GitHub savvy, you can fork and file a pull request for typos/edits. If you're not, you can file an issue.

Just be aware that this is not my primary job - I'll try to be as responsive as I can.

1 Object Based File Systems

Project storage on the DNAnexus platform is *object-based.* This kind of filesystem is probably very different from other file systems you've experienced.

The goal of this chapter is to show how the filesystem is different from your previous experiences, and to highlight ways to work successfully on the platform given this filesystem. Specifically you will be able to:

1. **Compare** and **contrast** differences between object-based file systems and POSIX filesystems
2. **Explain** the role of metadata and folders in organizing data and outputs
3. **Set up** a project for reproducible analysis
4. **Tag** and **utilize** tags for selecting files and separating output from multiple jobs.

1.1 Terminology

- **POSIX** - The filesystem that linux is based on. POSIX filesystems have *paths* and *files*.
- **Data** - The actual file contents. For example, for a CSV file, the actual header and rows correspond to the data.
- **Metadata** - information that is not part of the data, but is associated with the data. For our CSV file, some examples of metadata are the *file permissions* (who can access the file), the *creator*, and the *creation date*.
- **Object** - A contiguous set of memory that contains both the *data* and *metadata*. Has a unique identifier.
- **Unique Identifier** - the actual "address" for accessing the file. Unique to the file object when it is created. Does not change for the entire lifecycle of an object.
- **Database Engine** - software that allows for rapid searching and retrieving objects.

1.2 Review: POSIX Filesystems

You may be familar with *directory based* filesystems as a way to organize your data. The main way to find/refer to a file is through its *path*. What does this mean?

For example, if my file is called `chr1.vcf.gz` and it is located in the directory `/Geno_Data`, we'd refer to it using the *full path*:

```
/Geno_Data/chr1.vcf.gz
```

Paths are the main way we organize files for directory based systems. This information is external to a file. Most importantly, we use directories to organize and group files logically. For example, we might have our R script and the data it analyzes in the same directory.

For a file in a POSIX-based filesystem, **the path needs to be unique.** If they are not, there are rules for whether to replace that file with a new one, or to save both versions. For example, we can't have two files named:

```
/Geno_Data/chr1.vcf.gz
```

In the same folder. That violates our ability to find a file in the system.

https://grimoire.carcano.ch/blog/posix-compliant-filesystems/
https://www.kernel.org/doc/html/latest/filesystems/fuse.html

1.3 Object Based Filesystems are different

In contrast, object-based filesystems do not organize data like a folder based system. Each file object (such as a csv file, or a BAM file) has a unique identifier that identifies the data object. This unique identifier (like `file-aagejFHEJSEI`) serves as the main way to locate the data object, rather than the path.

However, file objects also have metadata that can be attached to them. This metadata can be:

Metadata Type	Example	Code Example
ID	`project-XXXXX:file-YYYYYY`	`dx mv file-YYYY raw_data/`
name	`chr1.tar.gz`	`dx find data --name chr1.tar.gz`
Path	`/raw_data/chr1.tar.gz`	
Creation Date		
Tags	`report`, `run1`	
Properties	`eid = 1153xxx`	

On the DNAnexus platform, both the *filename* and its path are considered **metadata**. This metadata is part of the object, along with the data portion of the object.

Importantly, *folders are not considered objects on the platform* - folders only exist within the metadata for the file objects.

The other issue is that the metadata for an object has no requirements to be unique. Which means you can have duplicates with the same file name in the same folder.

I know, this can be very distressing for most people. You can have two objects with the same file name, but they are considered distinct objects because they have unique identifiers.

https://www.ibm.com/cloud/blog/object-vs-file-vs-block-storage

1.4 Comparing POSIX and Object Based File Systems

Concept	POSIX File System	Object-Based System
File ID	Represented by Full Path	Represented by Object ID
Storage	Data with limited metadata	Metadata+Data
Path/Filename	Must be Unique	Can be duplicated
Metadata	Limited	Rich, can be freely modified

1.5 Tracing the journey of a file object onto the platform

When a file uploaded, file objects go through three stages before they are available. These stages are:

1. **Open** - Files are still being transferred via `dx upload` or the Upload Agent `ua`.
2. **Closing** - File objects stay in this state for no longer than 8-10 seconds.
3. **Closed** - Files are now available to be utilized on the platform, either with an app, workflow, or downloaded.

```
% dx describe file-FpQKQpQ0Fgk3gQZz3gPXQj7x
Result 1:
ID                      file-FpQKQpQ0Fgk3gQZz3gPXQj7x
Class                   file
Project                 project-GJ496B00FZfxPV5VG36FybvY
Folder                  /data
Name                    NA12878.bai
State                   closed
Visibility              visible
Types                   -
Properties              -
Tags                    -
Outgoing links          -
```

```
Created                 Wed Apr 22 17:59:22 2020
Created by              emiai
 via the job            job-FpQGX780FgkG4bGz86zZk04V
Last modified           Thu Oct 13 15:38:04 2022
Media type              application/octet-stream
archivalState           "live"
Size                    2.09 MB, sponsored by DNAnexus
cloudAccount            "cloudaccount-dnanexus"
```

1.6 Copying Files from One Project to Another

Copying has an important definition on the platform: *it means copying a file from one project to another project.* It doesn't refer to duplicating a file within a project.

When we copy a file from one project to another, a new physical copy is not made. The reference to the original file is copied. The data is identical and points to the same location on the platform, and the metadata is copied to the new project.

This is quite nice in that you are not doubly charged for storage on the platform. You do have the ability to clone files into a different project - that way a physical copy of the data is created.

Once the metadata is copied into the new project, there is no syncing of the metadata between the two projects. User 1 is free to modify the metadata in Project A and changes are not made to the metadata in Project B.

1.6.1 Advantages of Object Based Filesystems

The DNAnexus platform is only one example of an object-based filesystem. Other examples include Amazon S3 (Simple Storage Service), which is one of the biggest examples. Why does the world run on object based filesystems? There are a lot of advantages.

- **Highly scalable.** This is the main reason given for using an object-based system. Given that unique identifier, the data part of the object can be very large.
- **Fast Retrieval.** Object-based filesystems let us work with arbitrarily large file sizes, and we can actually stream files to and from workers.
- **Improved search speed.** You can attach various database engines to a set of objects and rapidly search through them. An example of such an engine is Snowflake.
- **File operations are simplified.** Compared to POSIX filesystems, there are only a few object filesystem commands: PUT, GET, DELETE, POST, HEAD.

1.6.2 Disadvantages of Object Based Filesystems

Coming from folder based filesystems, it can be a bit of a mind-bender getting used to object-based filesystems. Some of the disadvantages of Object Based Filesystems include:

- **Objects are immutable**. You can't edit an object in place. If you modify a file on a worker, you can't overwrite the original file object. A new file object must be created.
- **You have to be careful when generating outputs**. You can end up with two different objects with the same filename, and it can be some work to disambiguate these objects.
- **It's confusing.** You can actually have two files with the same filename in the same folder, because it is part of the changeable metadaa. Disambiguating these two files without using file-ids can be difficult. There are rules that govern this.
- **Metadata is much more important with file management.** Seriously, use tags for everything, including jobs and files. It will make working with multiple files much easier. And if you are on UKB RAP, leverage the file property metadata (`eid` and `field_id`) to help you select the files you want to process.

1.7 Always add project IDs to your File IDs

In Bash for Bioinformatics, we already discovered one way of working with files on the platform: `dx find data` (**?@sec-dx-find**). This is our main tool for selecting files with metadata.

When we work with files outside of our current project, we might reference it by a file-id. Using file IDs by themselves are a global operation and we need to be careful when we use this!

Why is this so? There is a search tree when we use a file ID that is not in our project and without a project context. The search over metadata is looking for a file object based on just file ID.

1. Look for the file in the current project
2. Look at all files across all other projects

If you want to use the platform effectively, you want to avoid 2. at all costs, especially when working with a lot of files. The metadata server will take much longer to find your files.

The lesson here is when using file-ids, it is safer to put the project-id in front of your file id such as below:

```
project-XXXXX:file-YYYYYYYYY
```

1.8 Batch Tagging

Remember, we can leverage `xargs` for tagging multiple files:

```
dx find data --name "*bam" --brief | xargs -I% sh -c "dx tag % 'bam'"
```

After we do this, we can check whether our operation was successful. We can run:

```
dx find data --tag bam --brief
```

And here is our response:

```
project-GJ496B00FZfxPV5VG36FybvY:file-BZ9YGzj0x05b66kqQv51011q
project-GJ496B00FZfxPV5VG36FybvY:file-BZ9YGpj0x05xKxZ42QPqZkJY
project-GJ496B00FZfxPV5VG36FybvY:file-BQbXVY0093Jk1KVY1J082y7v
project-GJ496B00FZfxPV5VG36FybvY:file-FpQKQk00FgkGV3Vb3jJ8xqGV
```

Using the `dx find data/xargs/dx tag` combination with various input parameters to `dx find data` such as `--name`, `--created-before`, `--created-after`, `--class`, will help us to batch tag files and other objects on the platform.

1.9 Use Case: Use tags for archiving

Let's do something concrete and useful in our project: tag files we no longer need for archiving.

Say there are files we want to archive. We can use `dx tag` or the DNAnexus UI to tag these files with a specific tag, such as `to_archive`. This can be done by users.

An administrator can then run a monthly job that archives the files with these tags using `dx api <project-id> archive --tag to_archive`.

What about dxFUSE?

You might ask about the role of dxFUSE with the Object Based Filesystem.
In short, dxFUSE makes the Object Based Filesystem of DNAnexus act like a POSIX filesystem. Specifically, if there are multiple objects with the same name within a folder, it provides a way to specify these objects using file paths.

Par-what-now?

You may have heard of *Parquet* files and wondered how they relate to file objects on the platform. They are a way to store data in a format called *columnar* storage.
It turns out it is faster for retrieval to store data not by *rows*, but by *columns*. This is really helpful because the data is sorted by *data* type and it's easier to traverse for this reason.
There are a number of *database engines* that are fast at searching and traversing these types of files. Some examples are Snowflake and Apache Arrow.
On the DNAnexus platform, certain data objects (called Datasets) are actually stored in Parquet format, for rapid searching and querying using Apache Spark.

2 Project and File Management

Successful project management at scale on the DNAnexus platform does require planning and though for how you will use the platform.

2.1 Learning Objectives

- **Explain** the role of organizations in project and member management
- **Utilize** different management strategies for administering your data to users
- **Describe** the roles that can be assigned to projects and how they interact with them
- **Annotate** your data with essential metadata such as tags and properties

2.2 Checklist for Successful Project Management

Based on past customers and how they interact with projects, I've tried to put together a checklist for new customers starting from scratch.

We assume that you will have multiple users, and that there is an administrator is in charge of the organization.

Task	Person Responsible
Creation of an Org	DNAnexus Support
Setting up billing for org	DNAnexus Support
Assigning an Admin for the Org	Org Admin
Enable Smart Reuse in Org	Org Admin
Creating a base project for incoming data	Org Admin

Task	Person Responsible
Uploading Data into base project	Org Admin/Uploader/Project Admin
Initial Portal Setup	xVantage Team
Enable Single Sign On	DNAnexus Team
Creating User Accounts for the Org	Org Admin
Creating projects/Copying Data from the base project	Project Admin
Setting Project Permissions (Downloads/Uploads/etc)	Project Admin
Assigning Users to projects with Roles	Project Admin

Org/Project Terminology

Here is a very useful link in case you get lost in terms of all of the terminology associated with projects and orgs: https://documentation.dnanexus.com/getting-started/key-concepts/organizations#glossary-of-org-terms

2.3 Starting Point: the Organization (or Org)

Before anything, you will need to create or have an organization (also known as an *org*) created for your group. This is usually done through contacting support (support@dnanexus.com).

What is an organization on the platform? It is an entity on the platform that is associated with multiple users. There are a lot of operations at the Org level that make effective data and project management possible.

Creating an org is important because an organization owns and administers the following:

- Control membership within your organization and access levels

- Set up for centralized Billing for your org
- Default app access level in the organization
- Default project access level in the organization
- Enable Smart Reuse for Organization
- Administer the portal for your own organization
- Access to files within projects administered by the org
- Creating an Audit Trail for the Organization

The Org Administrator has power over all projects and members in the Org. They are able to:

- Add and remove members to an org and change their access level
- Make themselves members of any project in the Org
- Change the Spending Limit of the Org
- Enable PHI Data Protections
- Enable Smart Reuse

2.4 Org-owned projects

In general, you will want to create projects within your org. This can be done by the org administrators. This simplifies billing for both file storage and compute.

Users can also create their own projects, but if they want to tie their project to the org's billing, they will need to transfer ownership to the organization.

PHI, Projects, and Orgs

There are many cases, especially when protected health information (PHI) is involved, where having a single organization doesn't make sense.
Controlling such data through its own organization may make much more sense for your group. For very large datasets, we recommend working with your Customer Success representative for implementation strategies.
Additionally, project admins can enable PHI Restrictions within a project.
Enabling an audit trail for the org also becomes very important, in order to be compliant with regulations such as 21 CFR Part 11. In short, an org admin will create a project for which the audit logs will be stored,

2.5 A Base Project: A Data Project for All Your Org's Data

One strategy for administering projects within your organization is to have a base project that contains all of your data (Figure 2.1). Tagging incoming data as they are uploaded can

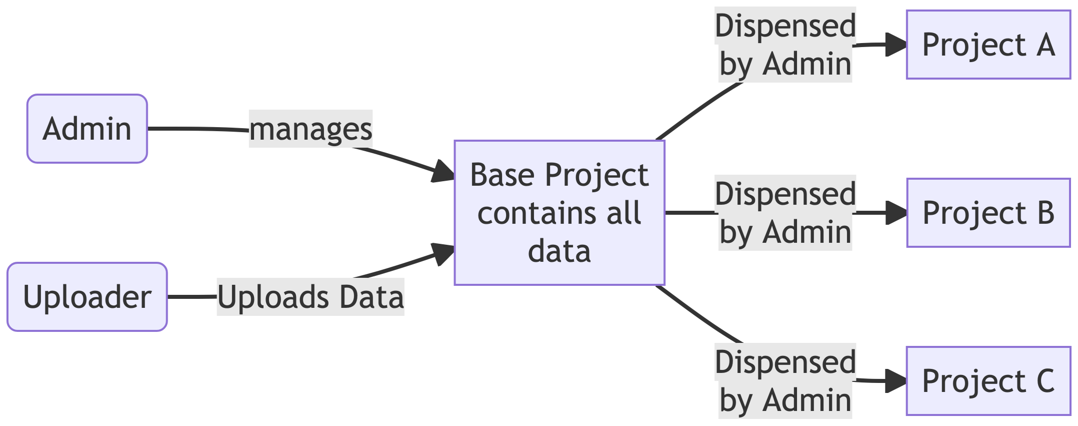


Figure 2.1: Using a base project to manage your Org's data. All other projects will be derived from this base project.

simplify project management. Then using these tags, you can copy the relevant data files to your separate projects.

The primary advantage of having a base project is that it allows for centralized data management. The org administrator / base project administrator controls access to all files within the org.

A related advantage of using base projects has to do with file deletion. Once a file is deleted in a project, it is not recoverable, unless they are also in the base project. Thus, having a base project can provide an overall safety net for the underlying files.

One final advantage of having a base project is that you can grant *upload access* to specific users to the base project. This is really helpful for when you have a sequencing group that needs to get raw data into your project.

2.6 Uploading Batch Files with the Upload Agent

Now that we've created a base project, we'll need to get our files into it. In our checklist, this can be done by the org admin, or an org user who has *uploader* access to the base project.

The DNAnexus Upload Agent software can be downloaded, installed, and used for automated batch uploading.

The Upload Agent is recommended over `dx upload` for large sets of files, because it is multi-threaded, and supports resuming in case an upload is interrupted. It will upload 1000 files at a time.

In particular, when uploading batches of files, such as everything that is in a folder, we recommend using the `--tag` or `--property` options to set metadata, such as tags.

2.7 Copying Files from One Project to Another

[Figure Here]

Copying has very specific definition on the DNAnexus platform: **it means copying a file from one project to another project.** Copying doesn't refer to duplicating a file within a project. You may be used to creating aliases or symbolic links within a project.

When we copy a file from one project to another, a new physical copy is not made. The reference to the original file is copied. The data is identical and points to the same location on the platform, and the metadata is copied to the new project.

This is quite nice in that you are not doubly charged for storage on the platform within your org. You do have the ability to clone files into a different project - that way a new physical copy of the data is created.

A Metadata Multiverse

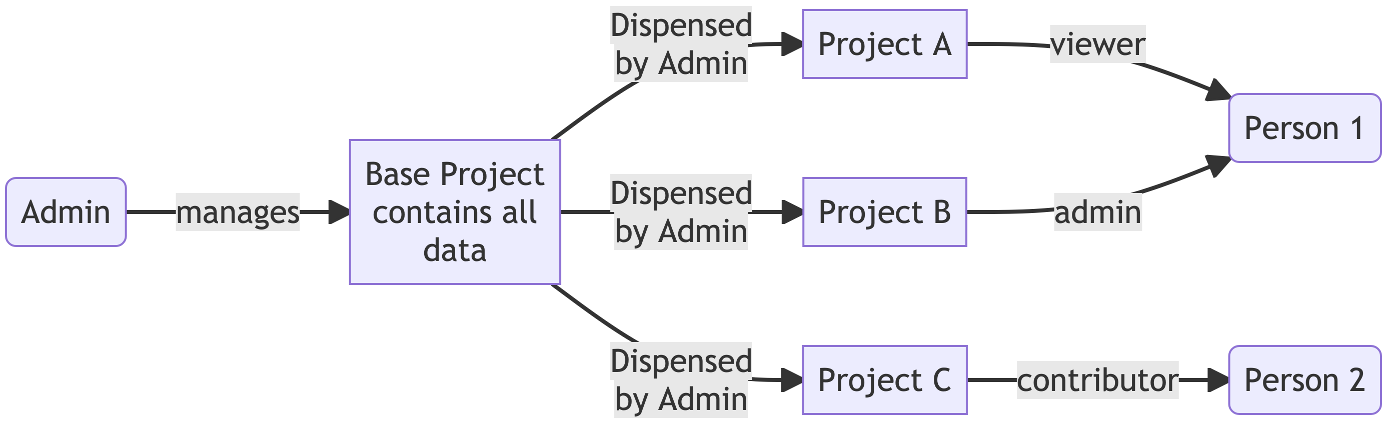
Once the metadata is copied into the new project, there is no syncing of the metadata between the two projects.
User 1 is free to modify the metadata in Project A and changes are not made to the metadata in Project B. However, the underlying data does not change.
Remember that the metadata for file objects includes file names, so you can actually change the file name in project B for the file object that you copied. This is not recommended, but it is possible.
Regardless, the underlying file id for the file object will remain the same.

Will I still be charged?

What happens when you have two copies of a file in two different org owned projects? What happens when one of these projects is deleted?
There is a process on the platform that is scans file-ids and whether they exist in a project for your org. If a reference to that file-id still exists in your project, then you will still be charged for it.
This is also why having a file archiving strategy is important when managing costs on

the platform.

2.8 Full Diagram



2.9 Some Rules of Thumb for Data Files in a Project

Given this structure, you will want to avoid having too many files within a project.

A good rule of thumb is to shoot for around 10,000 file objects total in a project. Going larger than this may impact the speed of your file searches.

Destructive Processes on the DNAnexus platform

There are certain operations on the platform that are destructive; that is, there is no ability to undo that operation:

- File / App / Workflow deletion
- Project deletion

Making your org users aware of these operations is important.
There is also the option to *disallow deletion* within a project to avoid issues like this. This is recommended especially in audit log projects to insure integrity of the audit logs.

3 Project Administration

In this section, we will talk about roles and what operations can be enabled/disabled in a project.

3.1 Project Level Roles

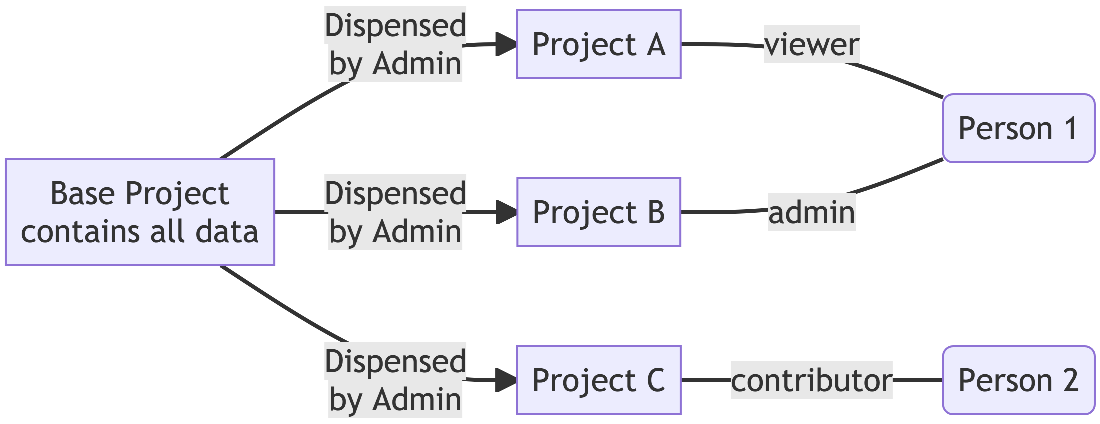


Figure 3.1: Possible roles within a project include *Admin*, *Contributor*, *Uploader*, and *Viewer*.

There are multiple roles that can be assigned to members of a project. In order of access (with each role inheriting privileges of the ones above):

Role	Description
Viewer	Can View Files and Apps within a Project
Uploader	Can upload files within a project; limited file management
Contributor	Can manage files and run apps/workflows
Admin	Top level - can manage membership and delete project

Project Administrators can also modify project-related flags for a project under the Settings Tab in a project. This includes:

- Enable PHI restrictions for a project, which supersedes any other flags set below
- Enable / Disallow file operations, including copying, uploading, and downloading
- Transfer Project Billing
- Project Deletion

3.2 A Suggested Project Structure

The DNAnexus platform is an object-based filesystem. That technically means that folders aren't needed. However, they are extremely helpful in helping you group your work.

For example, when I'm starting a project I usually have the following folder structure:

```
raw_data/    ## raw files
outputs/     ## processed Files
applets/     ## project-specific applets
workflows/.  ## project-specific workflows
notebooks/.  ## Jupyter Notebooks
```

4 Tags: Know Them, Love Them, Use Them

One of the ways to become a power user on the DNAnexus platform is to utilize tags when you generate output. This is important for reproducible analysis.

4.1 `dx tag`

Well, we've uploaded our files but forgot to tag them. We can apply tags to files using the `dx tag` command:

```
% dx tag file-FpQKQk00FgkGV3Vb3jJ8xqGV blah
```

If we do a `dx describe`:

```
% dx describe file-FpQKQk00FgkGV3Vb3jJ8xqGV
Result 1:
ID                    file-FpQKQk00FgkGV3Vb3jJ8xqGV
Tags:                 blah
```

4.2 Always Add a Project Context to your File IDs

We've already discovered one way of selecting multiple files on the platform: `dx find data` (sec-dx-find). This is our main tool for selecting files with metadata.

When we work with files outside of our current project, we might reference it by using a bare file-id. In general, we need to be careful when we use bare file IDs!

Why is this so? There is a search tree when we use a file ID that is not in our project and without a project context. The search over metadata is looking for a file object based on just file ID. Usually, when we provide a bare file ID, this search goes like the following:

1. Look for the file in the current project
2. Look at all files across all other projects (Very computationally expensive)

If you want to use the platform effectively, you want to avoid 2) at all costs, especially when working with a lot of files. The metadata server will take much longer to find your files.

The lesson here is *when using file-ids from a different project, it is much safer (and faster overall) to put the project-id in front of your file id* such as below:

```
project-XXXXX:file-YYYYYYYYY
```

5 Spark on the DNAnexus Platform

In this Chapter, we will discuss some of the intricacies behind working with Spark and on the DNAnexus platform.

I highly recommend that you go over the Cloud Computing Concepts chapter from Bash for Bioinformatics before you start tackling this.

As we'll discover, Spark is made for working with very large datasets. The datasets themselves might be very *long* (have billions of rows), or very *wide* (millions of columns).

A lot of datasets aren't this big. Especially with tools such as Apache Arrow, interactive analysis can handle data in the billion row range. Arrow support exists for both R (within the `DBI`/`tidyverse`/`arrow` packages), and Python (Pandas 2.0 now uses Arrow for its backend).

Genomic Data is often the exception for this. Whole Genome Sequencing outputs usually have hundreds of millions of variants, and with datasets such as UK Biobank, this is multiplied by a very large number of participants. We'll see in the next chapter that a framework exists for genomic analysis built on Spark called Hail.

Spark/Hail on The Core Platform

Please note that working with Spark requires an Apollo License if you are on the Core Platform. For more information, please contact DNAnexus sales.
If you are on UKB RAP, the Pheno Data is stored as a Spark Database, so you do not have to have a license on RAP. You also have access to the Hail configurations of Spark JupyterLab as well.

5.1 Spark Concepts

In this section, we will discuss the basic concepts that you need to know before using Spark successfully on the platform.

On the platform, Spark is used to work with ingested Pheno Data either indirectly using the `dx extract_dataset` functionality, or by working directly with the Spark Database using Libraries such as `PySpark` (for Python) or `sparklyr` (for R), or using `Spark SQL` to query the contents of a database.

Par-what-now?

You may have heard of *Parquet* files and wondered how they relate to database objects on the platform. They are a way to store data in a format called *columnar* storage.
It turns out it is faster for retrieval and searching to store data not by *rows*, but by *columns*. This is really helpful because the data is sorted by *data* type and it's easier to traverse for this reason.
Apache Spark is the scalable solution to using columnar formats such as Parquet. There are a number of *database engines* that are fast at searching and traversing these types of files. Some examples are Snowflake and Apache Arrow.
On the DNAnexus platform, certain data objects (called Dataset and Database objects) are actually stored in Parquet format, for rapid searching and querying using Apache Spark.

5.1.1 What is Spark?

Apache Spark is a framework for working with large datasets that won't fit into memory on a single computer. Instead, we use what's called a Spark Cluster. Specifically, we request the Spark Cluster as part of a Spark JupyterLab configuration.

The main pieces of a Spark Cluster are the *Driver Node*, and the *executors*, which correspond to individual cores on a *Worker Node*. Individual worker nodes have processes known as *Executors* that will run on their portion of the data.

Spark lets us analyse large datasets by *partitioning* the data into smaller pieces that are made available to the cluster via the Hadoop File System. Each executor gains access to a piece of the data, and executes *tasks* assigned to it on this piece of the data.

The Driver Node directs the individual executors by assigning *tasks*. These individual tasks are part of an overall execution plan.

This link is a good introduction to Spark Terminology.

5.2 Spark and Lazy Evaluation

One thing that is extremely important to know is that Spark executes operations in a *Lazy* manner. This is opposed to something like R or Python which executes commands right away (a *greedy* manner).

When we run Spark operations, we are usually building chains of operations.

5.3 The Four Filesystems

Using Spark on DNAnexus effectively requires us to be familiar with 4 different file systems (yes, you heard that right):

- **dxFUSE** file system - used to directly access project storage. We access this using `file:///mnt/project/` in our scripts. For example, if we need to access a file named `chr1.tar.gz`, we refer to it as `file:///mnt/project/chr1.tar.gz`. Mounted at the start of JupyterLab; to access
- **Driver Node** File System - our waypoint for getting things into and out of the Hadoop File System and possibly into Project Storage. Since we run JupyterLab on the Driver Node, we use `hdfs://` to transfer from the Driver Node to the Hadoop File system (see below), and use `dxFUSE` to work with Project Storage files, and `dx upload` to transfer files back into project storage.
- **Hadoop** File System (HDFS) - Distributed File System for the cluster. We access this using the `hdfs://` protocol.
- **dnax** S3 Bucket - contains ingested data. This is a separate S3 bucket. We interact with ingested Pheno and Geno Data using a special protocol called `dnax://`. Ingested data that is stored here is accessed through connecting to the DNAX database that holds it. Part of why Cohort Browser works with very large data is that the ingested data is stored in this bucket.

The TL;DR (too long; didn't read) to this all is:

1. Try to avoid transfers through the Driver node (`hdfs dfs -get`/`dx upload`), as you are limited to the memory/disk space of the Driver node.
2. When possible, use dxFUSE with glob wildcards to directly stream files into HDFS (Spark Cluster Storage) from project storage by using `file:///mnt/project/` URLs. For example, we could load a set of pVCF files with `hl.read_vcf(file:///mnt/project/data/pVCF/*.vcf.gz)`)
3. If you need to get files out of HDFS into Project Storage, use dxFUSE in `-limitedWrite` mode to write into project storage by passing in `file:///mnt/project/` URLs to the `.write()` methods in Spark/Hail. You will need to remount dxFUSE in the JupyterLab container to do this.
4. If you need to persist your Spark DataFrame, use `dnax:///<db_name>/<table_name>` with `.write()` to write it into DNAX storage (). Then use SparkSQL to save it in the Spark Database in DNAX.

Accessing Spark Databases on DNAnexus

One issue I see a lot of people get tripped up with is with Database Access. One thing to know is that Spark databases cannot be moved from project to project. Once the data is ingested into a database object, it can't be copied to a new project.
For a user to access the data in a Spark Database, they must have at least

VIEW level access to the project that contains it on the platform.
Note that *Datasets* (not *Databases*) can be copied to new projects. However, to actually access the data in Cohort Browser, the project that contains the database needs to be viewable by the user.

Spark Cluster JupyterLab vs. Spark Apps

Our main way of using Spark on DNAnexus is using the Spark JupyterLab app. This app will let you specify the instance types used in the Spark Cluster, the number of nodes in your Spark cluster, as well as additional features such as Hail or Variant Effect Predictor. This should be your main way of working with Spark DataFrames that have been ingested into Apollo.
You can also build Spark Applets that request a Spark Cluster. This is a more complicated venture, as it may require installing software on the worker nodes using a bootstrapping script, distributing data using `hdfs dfs -put`, and submitting Spark Jobs using `dx-spark-submit`.

5.4 Balancing Cores and Memory

Central to both Spark and Hail is tuning the number of cores and memory in each of our instances. This also needs to be balanced with the partition size for our data. Data can be repartitioned, although this may be an expensive operation.

In general, the number of data partitions should be greater than the number of cores. Why? If there are less partitions than cores, then the remaining cores will be unused.

There are a lot of other tuning considerations to keep in mind.

5.5 The Spark UI

When you start up Spark JupyterLab, you will see that it will open at a particular url which has the format `https://JOB-ID.dnanexus.com`. To open the Spark UI, use that same URL, but with a `:8081` at the end of it. For example, the UI url would be `https://JOB-ID.dnanexus.com:8081`.

Note that the Spark UI is not launched until you connect to Spark or Hail.

The Spark UI is really helpful in understanding how Spark translates a series of operations into a Plan, and how it executes that plan.

5.6 Extracting PhenoData with `dx extract_dataset`

First of all, you may not need to use Spark JupyterLab if you just want to extract Pheno Data from a Dataset and you know the fields.

There is a utility within dx-toolkit called `dx extract_dataset` that will retrieve pheno data in the form of a CSV. It uses Spark, but the Spark is executed by the Thrift Server (the same Spark Cluster that serves the data for Cohort Browser).

You will need a list of field ids to retrieve them.

However, there are good reasons to work with the Spark Database directly. Number one is that your Cohort query will not run on the Thrift Server. If your query is complex and takes two minutes to execute, the Thrift Server will timeout. In that case, you need to run your own Spark JupyterLab cluster and extract the data using Spark itself.

5.7 Connecting with `PySpark` (For Python Users)

The first thing we'll do to connect to the Spark database is connect to it by starting a Spark Session. Make sure you only do this once. If you try to connect twice, Spark will throw an error.

If this happens, make sure to restart your notebook kernel.

~~~
```{python}
import pyspark
sc = pyspark.SparkContext()
spark = pyspark.sql.SparkSession(sc)
```
~~~

5.7.1 Running SparkSQL Queries in PySpark

The basic template for running Spark Queries is this:

~~~
```{python}
retrieve_sql = 'select .... from .... '
df = spark.sql(retrieve_sql)
```
~~~

5.7.2 Koalas is the Pandas version of Spark

If you are familiar with Pandas, the Koalas module (from Databricks) provides a Pandas-like interface to SparkSQL queries. This way, you don't have to execute native Spark commands or SparkSQL queries.

Once you have a Spark DataFrame, you can convert it to a Koalas one by using the `.to_koalas()` method.

~~~
```{python}
df_koalas = df.to_koalas()
```
~~~

5.8 Connecting with `sparklyr` (For R Users)

You'll need to install the package **`sparklyr`** along with its dependencies to work with the Spark DataFrames directly with R.

~~~
```{r}
library(DBI)
library(sparklyr)
port <- Sys.getenv("SPARK_MASTER_PORT")
master <- paste("spark://master:", port, sep = '')
sc = spark_connect(master)
```
~~~

~~~
```{r}
retrieve_sql <- 'select .... from .... '
df = dbGetQuery(sc, retrieve_sql)
```
~~~

6 Hail on DNAnexus

After reading this chapter, you should be able to:

- **Explain** basic Hail terms and concepts
- **Import data** into a Hail MatrixTable from pVCF and BGEN formats
- **Merge** phenotypic data into the MatrixTable
- **QC** and **Filter** Variant data for use in a GWAS using Hail
- **QC** and **Filter** Sample data for use in GWAS with Hail
- **Run** GWAS on a Hail MatrixTable
- **Annotate** Hail MatrixTables with internal DNAnexus resources
- **Export** results from Hail to CSV format

6.1 What is Hail?

Hail is a genomics framework built on top of Spark to allow for scalable genomics queries. It uses many of the same concepts (partitions, executors, plans) but it is genomics focused.

For example, the Hail MatrixFrame can be QC'ed, and then filtered on these QC metrics.

6.2 Vocabulary

DNAnexus Specific:

- **DNAX** - S3 bucket for storing Apollo Databases and Tables - accessed with the `dnax://` protocol
- **Database Object** - data object on platform that represents a parquet database - has a unique identifier.

Hail Specific Terminology:

- **Spark Driver** - the "boss" of the Spark cluster - hosts the Spark UI
- **Spark Worker** - individual nodes that house executors
- **Hail Executor** - individual cores/CPUs within a worker - Executors are assigned tasks, identical to Spark.

- **Data Partition** - portion of the data that is accessed by worker in parquet columnar format
- **Key** - unique identifier for rows or columns
  - row keys: *rsid + alleles*
  - column keys: *sample IDs*

6.3 Very first thing: Connect to Hail

The first thing we'll do is connect to Hail. In our Spark DXJupyter Instance, we'll use the following boilerplate code:

```
# Running this cell will output a red-colored message- this is expected.
# The 'Welcome to Hail' message in the output will indicate that Hail is ready to use in the

from pyspark.sql import SparkSession
import hail as hl

builder = (
    SparkSession
    .builder
    .enableHiveSupport()
)
spark = builder.getOrCreate()
hl.init(sc=spark.sparkContext)
```

This will be the response from Hail:

```
pip-installed Hail requires additional configuration options in Spark referring
  to the path to the Hail Python module directory HAIL_DIR,
  e.g. /path/to/python/site-packages/hail:
    spark.jars=HAIL_DIR/hail-all-spark.jar
    spark.driver.extraClassPath=HAIL_DIR/hail-all-spark.jar
    spark.executor.extraClassPath=./hail-all-spark.jarRunning on Apache Spark version 2.4.4
SparkUI available at http://ip-172-31-34-162.ec2.internal:8081
Welcome to
     __  __     <>__
    / /_/ /__  __/ /
   / __  / _ `/ / /
  /_/ /_/\_,_/_/_/   version 0.2.78-b17627756568
LOGGING: writing to /opt/notebooks/hail-20230418-1956-0.2.78-b17627756568.log
```

Now we are ready to connect and do work with Hail.

Note that you should only initialize one time in a notebook. If you try to initialize multiple times, then you will get an error.

If that happens, restart the notebook kernel (**Notebook** > **Restart Kernel** in the menu) and start executing all over again.

6.4 Important Data Structures in Hail: Tables and MatrixTables

Before we get started with doing a GWAS in Hail, let's talk about the data structures we'll work with. *Hail Tables* and *Hail MatrixTables*.

One of the hardest things to understand about Hail data structures is that they are actually linked data structures. It's easiest to think of them as data tables with attached metadata.

6.5 Hail Table

The fundamental data structure in Hail is the Hail Table.

I think of Hail Tables as a regular data table with a set of metadata fields. These metadata fields are known as **global fields**. You can think of them as annotations for the entire table, such as the instrument used to generate the table, the software version used to process the data, etc.

There are multiple operations we can do on a Hail Table (let's call our Hail Table as `ht` here):

- `ht.filter()`
- `ht.annotate()`
- `ht.show()`

Let's look at some examples of these operations. We can find a list of the row fields for table using `.describe()`:

```
ht.describe()
```

As we work with MatrixTables, we'll see that it is actually a little conceptually easier to filter on our sample QC data, then join it back to the MatrixTable to filter out samples that don't meet our criteria.

6.6 Hail MatrixTable

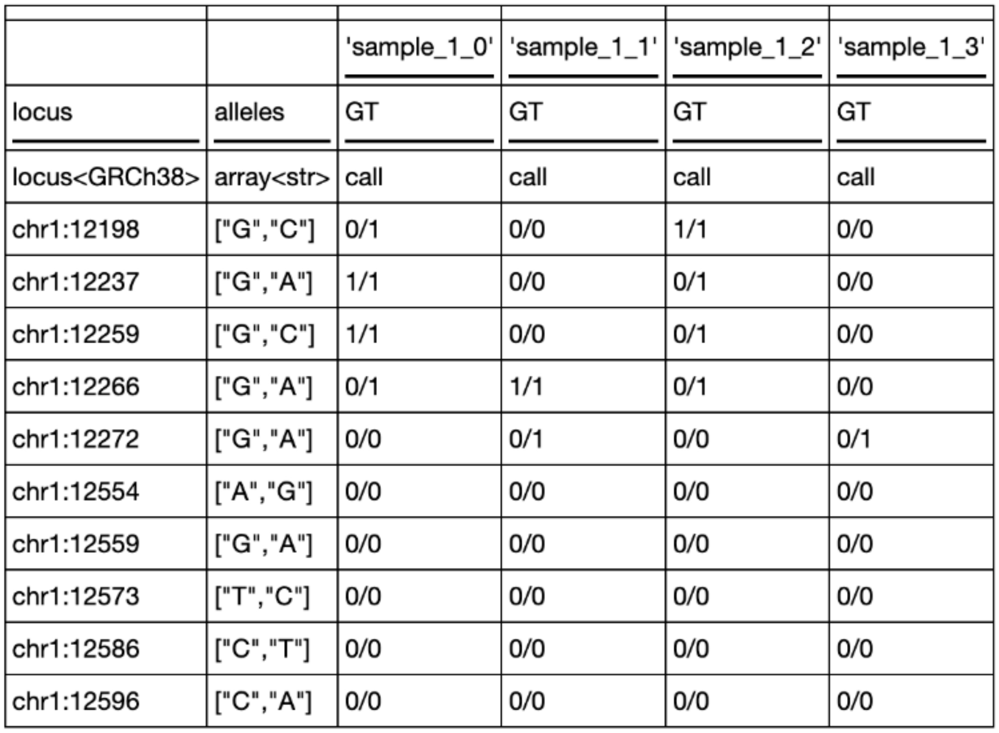
In contrast, Hail MatrixTables consist of the entries (genotypes) that have metadata attached to both the rows and columns. (They also have global fields, but let's ignore that for now.)

The main data structure is the table itself, which contains the genotypes. This table is considered as the **entry fields**. Each entry (or cell) contains data.

For example, if we call

```
mt.GT.show()
```

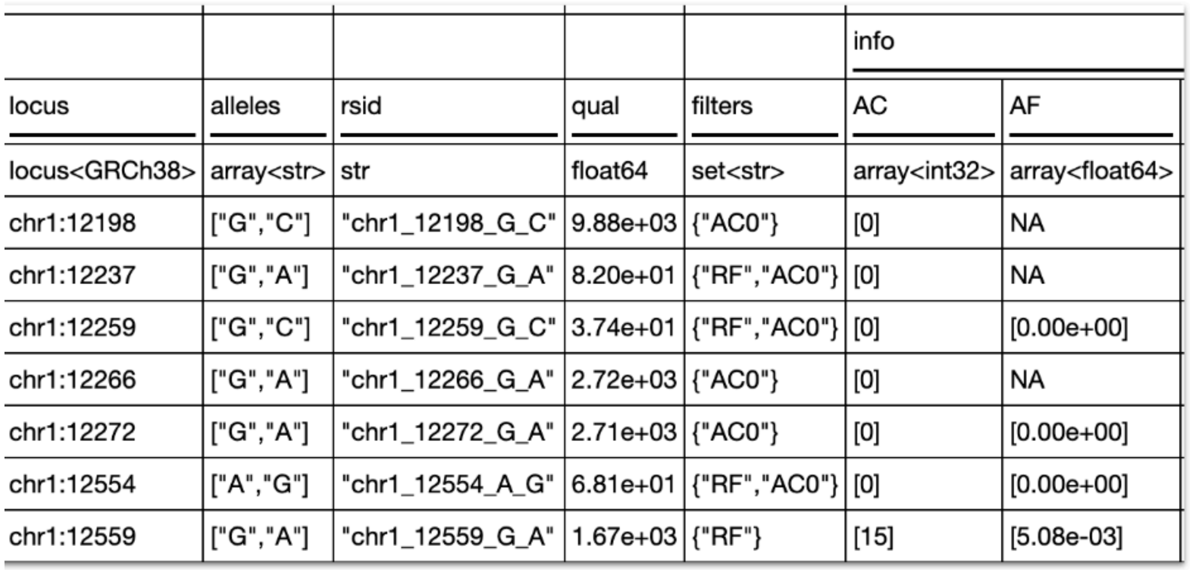
It will show the first few rows and columns of the genotypes (the example shown is synthetic data):



		'sample_1_0'	'sample_1_1'	'sample_1_2'	'sample_1_3'
locus	alleles	GT	GT	GT	GT
locus<GRCh38>	array<str>	call	call	call	call
chr1:12198	["G","C"]	0/1	0/0	1/1	0/0
chr1:12237	["G","A"]	1/1	0/0	0/1	0/0
chr1:12259	["G","C"]	1/1	0/0	0/1	0/0
chr1:12266	["G","A"]	0/1	1/1	0/1	0/0
chr1:12272	["G","A"]	0/0	0/1	0/0	0/1
chr1:12554	["A","G"]	0/0	0/0	0/0	0/0
chr1:12559	["G","A"]	0/0	0/0	0/0	0/0
chr1:12573	["T","C"]	0/0	0/0	0/0	0/0
chr1:12586	["C","T"]	0/0	0/0	0/0	0/0
chr1:12596	["C","A"]	0/0	0/0	0/0	0/0

We call the row metadata **row fields**, which correspond to the genomic variants in the MatrixTable. *Operations on the row fields are operations on genomic variants.* (technically the data in a Hail Table are also called row fields, but I find this to be confusing).

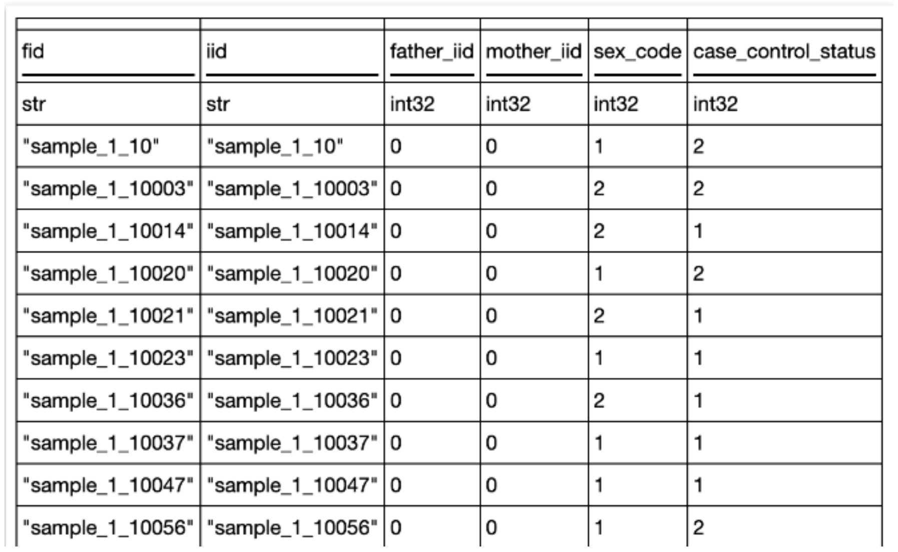
Here's an example of some row fields. We'll see that the rows correspond to variants, and the columns correspond to metrics or annotations about the variants.



					info	
locus	alleles	rsid	qual	filters	AC	AF
locus<GRCh38>	array<str>	str	float64	set<str>	array<int32>	array<float64>
chr1:12198	["G","C"]	"chr1_12198_G_C"	9.88e+03	{"AC0"}	[0]	NA
chr1:12237	["G","A"]	"chr1_12237_G_A"	8.20e+01	{"RF","AC0"}	[0]	NA
chr1:12259	["G","C"]	"chr1_12259_G_C"	3.74e+01	{"RF","AC0"}	[0]	[0.00e+00]
chr1:12266	["G","A"]	"chr1_12266_G_A"	2.72e+03	{"AC0"}	[0]	NA
chr1:12272	["G","A"]	"chr1_12272_G_A"	2.71e+03	{"AC0"}	[0]	[0.00e+00]
chr1:12554	["A","G"]	"chr1_12554_A_G"	6.81e+01	{"RF","AC0"}	[0]	[0.00e+00]
chr1:12559	["G","A"]	"chr1_12559_G_A"	1.67e+03	{"RF"}	[15]	[5.08e-03]

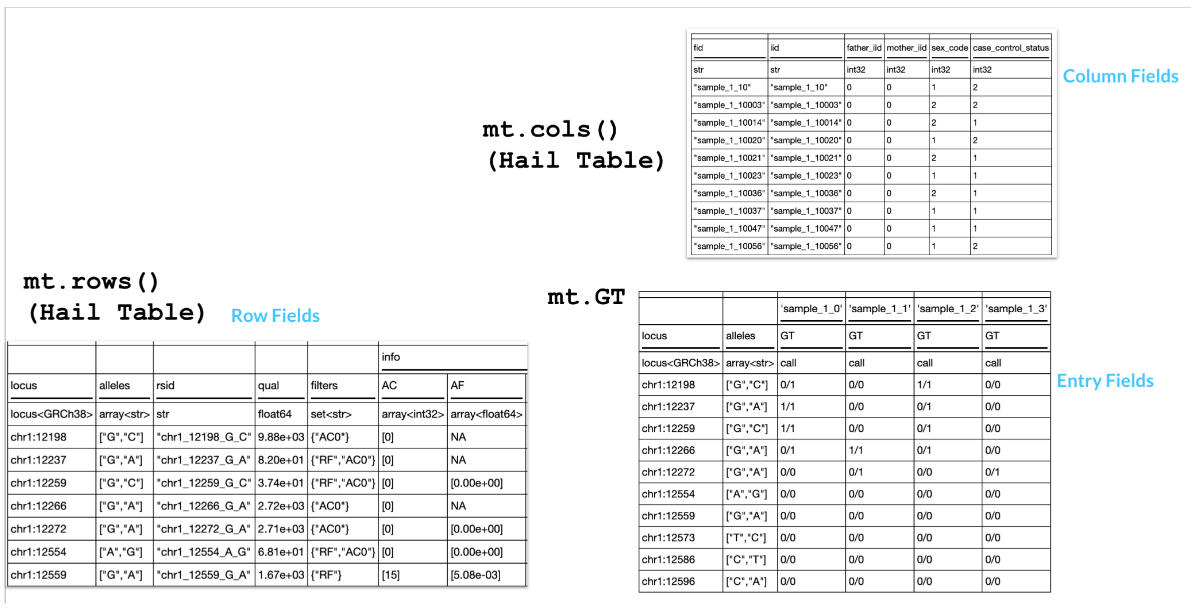
We call the column metadata **column fields**, which correspond to the samples in the MatrixTable. The rows in this table correspond to samples in our MatrixTable, and the columns correspond to per sample covariates (such as gender, age).

Here's an example of some column fields.



fid	iid	father_iid	mother_iid	sex_code	case_control_status
str	str	int32	int32	int32	int32
"sample_1_10"	"sample_1_10"	0	0	1	2
"sample_1_10003"	"sample_1_10003"	0	0	2	2
"sample_1_10014"	"sample_1_10014"	0	0	2	1
"sample_1_10020"	"sample_1_10020"	0	0	1	2
"sample_1_10021"	"sample_1_10021"	0	0	2	1
"sample_1_10023"	"sample_1_10023"	0	0	1	1
"sample_1_10036"	"sample_1_10036"	0	0	2	1
"sample_1_10037"	"sample_1_10037"	0	0	1	1
"sample_1_10047"	"sample_1_10047"	0	0	1	1
"sample_1_10056"	"sample_1_10056"	0	0	1	2

Putting everything together, we see this:



fid	iid	father_iid	mother_iid	sex_code	case_control_status
str	str	int32	int32	int32	int32
"sample_1_10"	"sample_1_10"	0	0	1	2
"sample_1_10003"	"sample_1_10003"	0	0	2	2
"sample_1_10014"	"sample_1_10014"	0	0	2	1
"sample_1_10020"	"sample_1_10020"	0	0	1	2
"sample_1_10021"	"sample_1_10021"	0	0	2	1
"sample_1_10023"	"sample_1_10023"	0	0	1	1
"sample_1_10036"	"sample_1_10036"	0	0	2	1
"sample_1_10037"	"sample_1_10037"	0	0	1	1
"sample_1_10047"	"sample_1_10047"	0	0	1	1
"sample_1_10056"	"sample_1_10056"	0	0	1	2

					info	
locus	alleles	rsid	qual	filters	AC	AF
locus<GRCh38>	array<str>	str	float64	set<str>	array<int32>	array<float64>
chr1:12198	["G","C"]	"chr1_12198_G_C"	9.88e+03	{"AC0"}	[0]	NA
chr1:12237	["G","A"]	"chr1_12237_G_A"	8.20e+01	{"RF","AC0"}	[0]	NA
chr1:12259	["G","C"]	"chr1_12259_G_C"	3.74e+01	{"RF","AC0"}	[0]	[0.00e+00]
chr1:12266	["G","A"]	"chr1_12266_G_A"	2.72e+03	{"AC0"}	[0]	NA
chr1:12272	["G","A"]	"chr1_12272_G_A"	2.71e+03	{"AC0"}	[0]	[0.00e+00]
chr1:12554	["A","G"]	"chr1_12554_A_G"	6.81e+01	{"RF","AC0"}	[0]	[0.00e+00]
chr1:12559	["G","A"]	"chr1_12559_G_A"	1.67e+03	{"RF"}	[15]	[5.08e-03]

		'sample_1_0'	'sample_1_1'	'sample_1_2'	'sample_1_3'
locus	alleles	GT	GT	GT	GT
locus<GRCh38>	array<str>	call	call	call	call
chr1:12198	["G","C"]	0/1	0/0	1/1	0/0
chr1:12237	["G","A"]	1/1	0/0	0/1	0/0
chr1:12259	["G","C"]	1/1	0/0	0/1	0/0
chr1:12266	["G","A"]	0/1	1/1	0/1	0/0
chr1:12272	["G","A"]	0/0	0/1	0/0	0/1
chr1:12554	["A","G"]	0/0	0/0	0/0	0/0
chr1:12559	["G","A"]	0/0	0/0	0/0	0/0
chr1:12573	["T","C"]	0/0	0/0	0/0	0/0
chr1:12586	["C","T"]	0/0	0/0	0/0	0/0
chr1:12596	["C","A"]	0/0	0/0	0/0	0/0

6.7 Keep in Mind

The cardinal rule of MatrixTables is this:

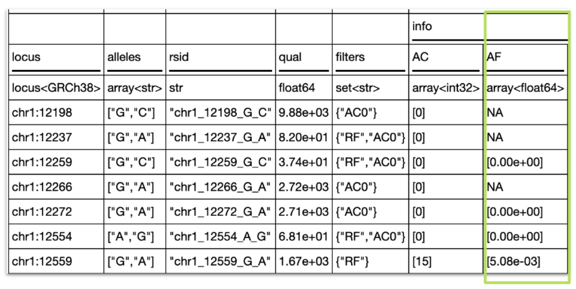
> **Row** operations operate on *variants*, while **Column** operations operate on *samples*.

It is easy to get lost in a chain of Hail commands and not be able to understand what the code is doing unless you concentrate on this.

Rows = Variants

Operations alter variants (SNPs, etc)

```
new_mt = mt.filter_rows(mt.info.AF > 0.9)
```

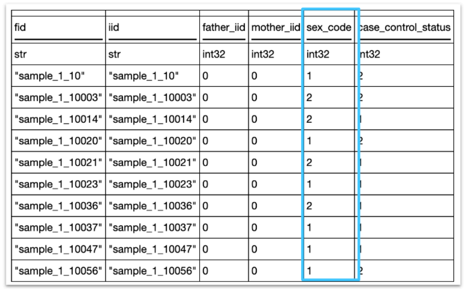


					info	
locus	alleles	rsid	qual	filters	AC	AF
locus<GRCh38>	array<str>	str	float64	set<str>	array<int32>	array<float64>
chr1:12198	["G","C"]	"chr1_12198_G_C"	9.88e+03	{"AC0"}	[0]	NA
chr1:12237	["G","A"]	"chr1_12237_G_A"	8.20e+01	{"RF","AC0"}	[0]	NA
chr1:12259	["G","C"]	"chr1_12259_G_C"	3.74e+01	{"RF","AC0"}	[0]	[0.00e+00]
chr1:12266	["G","A"]	"chr1_12266_G_A"	2.72e+03	{"AC0"}	[0]	NA
chr1:12272	["G","A"]	"chr1_12272_G_A"	2.71e+03	{"AC0"}	[0]	[0.00e+00]
chr1:12554	["A","G"]	"chr1_12554_A_G"	6.81e+01	{"RF","AC0"}	[0]	[0.00e+00]
chr1:12559	["G","A"]	"chr1_12559_G_A"	1.67e+03	{"RF"}	[15]	[5.08e-03]

Columns = Samples

Operations alter samples (participants)

```
new_mt = mt.filter_cols(mt.sex_code == 1)
```



fid	iid	father_iid	mother_iid	sex_code	case_control_status
str	str	int32	int32	int32	int32
"sample_1_10"	"sample_1_10"	0	0	1	2
"sample_1_10003"	"sample_1_10003"	0	0	2	2
"sample_1_10014"	"sample_1_10014"	0	0	2	1
"sample_1_10020"	"sample_1_10020"	0	0	1	2
"sample_1_10021"	"sample_1_10021"	0	0	2	1
"sample_1_10023"	"sample_1_10023"	0	0	1	1
"sample_1_10036"	"sample_1_10036"	0	0	2	1
"sample_1_10037"	"sample_1_10037"	0	0	1	1
"sample_1_10047"	"sample_1_10047"	0	0	1	1
"sample_1_10056"	"sample_1_10056"	0	0	1	2

Storing MatrixTables in DNAX

In general, if you are loading from pVCF files, we suggest that you save the MatrixTable into DNAX. This is because loading pVCF files is an expensive operation compared to BGEN files.
However, we suggest doing filtering and QC of your variants before you save your MatrixTable to DNAX. In general, you will be doubly charged for MatrixTable storage along with your pVCF file storage if you store without filtering, so it is worth reducing the data stored in DNAX.
Note that BGEN files can be directly loaded from Project Storage, but if there is a large amount of filtering/preprocessing involved, it is also worth storing the results as a MatrixTable in DNAX for others to use/access downstream.

6.8

7 GWAS using Hail

This table shows the basic GWAS process using Hail. We will cover each of these in the sections below.

Step	Description	Code Example
0.	Initiate Spark and Hail	See Notebook
1.	Load pVCF/BGEN data, save as MatrixTable (MT)	`mt = hl.import_vcf(path)` or `mt = hl.import_bgen(path)`
2.	Load Pheno file, merge with MatrixTable	`phenogeno_mt = mt.annotate_cols(**pheno_table[mt.s])`
3.	Build Sample QC table from MT, use to filter MT	`sample_qc_tb = hl.sample_qc(mt); qc_mt = phenogeno_mt.semi_join_rows(locus_qc_tb`
4.	Build Locus QC table from MT, use to filter MT	`locus_qc_tb = hl.locus_qc(mt); qc_mt = qc_mt.semi_join_cols(sample_qc_tb)`
5.	Run GWAS	`gwas_tb = hl.logistic_regression.rows()`
6.	Visualize Results	`manhattan_plot = hl.plot.manhattan(gwas_tb.p_value); show(manhattan_plot)`
7.	Annotate Results with VEP or annotation db	`ann_gwas_tb = db.annotate_rows_db(gwas_tb, 'gencode'`
8.	Save results to CSV, export chromosomes as BGEN file	See Notebook

References