DA5020 Spring 2017: Scraping, Parsing, and Storing Illumina Sample Sheets from AWS S3

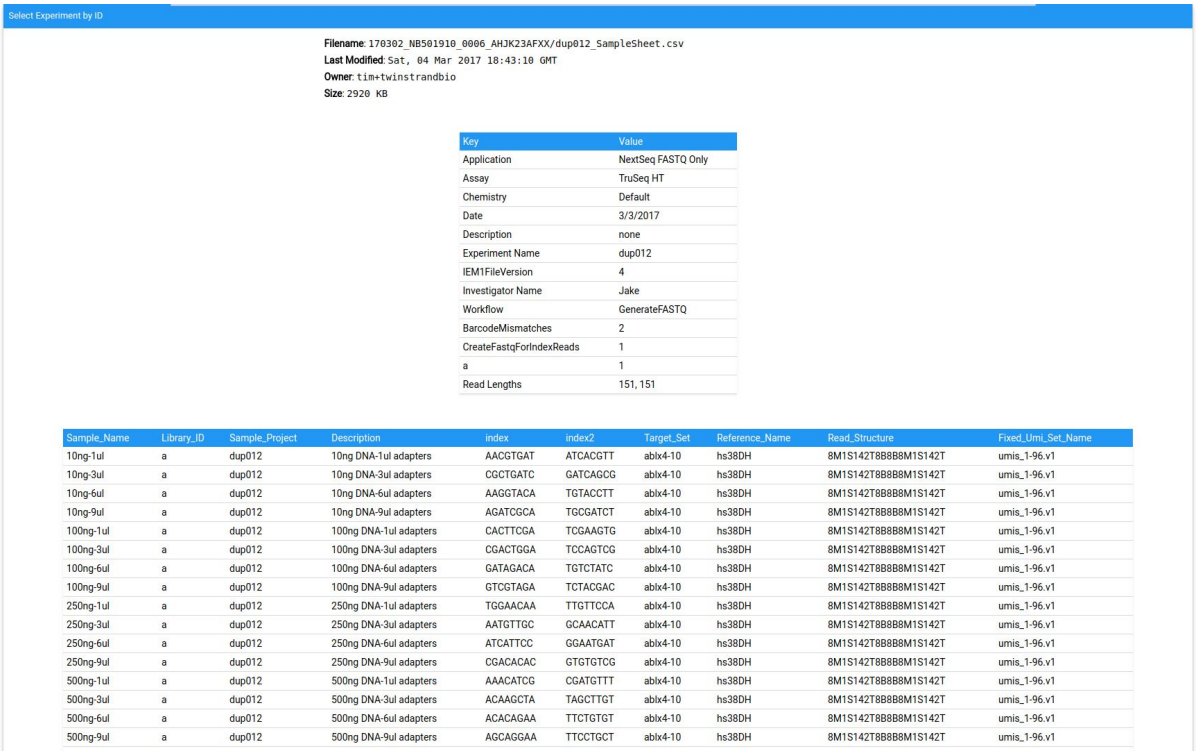
Clint Valentine

April 20, 2017

Introduction

Many startups in the field of next-generation sequencing are faced with the immediate need to track metadata, metrics, and progress reports of experiments. Most laboratory information management systems are not well suited for bespoke tasks that computational biologists often need from such a data collection. This need stems purely from innovation. My colleagues and I are a team of researchers at the fringe of custom applications in next-generation sequencing and many of the technologies we adapt and build need to be tuned and monitored through many time-scales. We have identified a need for a data store designed to tightly integrate with analytical tools we build in house. A simple interface for interacting with this data is essential for every member of our small team and a programmatic interface will prove most valuable for those of us that work specifically with analysing trends in data over time.

This project summarizes a solution to the need for a centralized data store and uses the technologies `R`, `AWS S3`, the `AWS S3 SDK`, `SQLite`, `Python`, and web programming languages such as `HTML` and `CSS`.



Key	Value
Application	NextSeq FASTQ Only
Assay	TruSeq HT
Chemistry	Default
Date	3/3/2017
Description	none
Experiment Name	dup012
IEM1FileVersion	4
Investigator Name	Jake
Workflow	GenerateFASTQ
BarcodeMismatches	2
CreateFastqForIndexReads	1
a	1
Read Lengths	151, 151

Sample_Name	Library_ID	Sample_Project	Description	index	index2	Target_Set	Reference_Name	Read_Structure	Fixed_Umi_Set_Name
10ng-1ul	a	dup012	10ng DNA-1ul adapters	AACGTGAT	ATCACGTT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
10ng-3ul	a	dup012	10ng DNA-3ul adapters	CGCTGATC	GATCAGCG	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
10ng-6ul	a	dup012	10ng DNA-6ul adapters	AAGGTACA	TGTACCTT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
10ng-9ul	a	dup012	10ng DNA-9ul adapters	AGATCGCA	TGCGATCT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
100ng-1ul	a	dup012	100ng DNA-1ul adapters	CACTTCGA	TCGAAGTG	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
100ng-3ul	a	dup012	100ng DNA-3ul adapters	CGACTGGA	TCCAGTCG	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
100ng-6ul	a	dup012	100ng DNA-6ul adapters	GATAGACA	TGTCTATC	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
100ng-9ul	a	dup012	100ng DNA-9ul adapters	GTCGTAGA	TCTACGAC	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
250ng-1ul	a	dup012	250ng DNA-1ul adapters	TGGAACAA	TTGTTCCA	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
250ng-3ul	a	dup012	250ng DNA-3ul adapters	AATGTTGC	GCAACATT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
250ng-6ul	a	dup012	250ng DNA-6ul adapters	ATCATTCC	GGAATGAT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
250ng-9ul	a	dup012	250ng DNA-9ul adapters	CGACACAC	GTGTGTCG	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
500ng-1ul	a	dup012	500ng DNA-1ul adapters	AAACATCG	CGATGTTT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
500ng-3ul	a	dup012	500ng DNA-3ul adapters	ACAAGCTA	TAGCTTGT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
500ng-6ul	a	dup012	500ng DNA-6ul adapters	ACACAGAA	TTCTGTGT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1
500ng-9ul	a	dup012	500ng DNA-9ul adapters	AGCAGGAA	TTCCTGCT	ablx4-10	hs38DH	8M1S142T8B8B8M1S142T	umis_1-96.v1

Figure 1: A screenshot of the web app which queries the SQLite database if provided an experiment identifier. An experiment identifier is not an identifier that can faithfully represent a sequencing run since many experiments can be multiplexed on an Illumina sequencer. There exists a button in the header menu which pops up an overlay where a user can type in a case-insensitive, sanitized, whitespace-stripped experiment identifier which then loads into view the results (as shown in screenshot). This is the ultimate functional representation of the project.

Project Design and Issues

Our existing infrastructure for long term data storage exists on `AWS s3` and has been setup as a remote repository for all of our Illumina sequencing output. Data from sequencing machines is synced in realtime to dedicated run folders with a unique naming scheme. This data is then analyzed via `AWS EC2` instances and then synced back to `AWS S3`. Once our informatic pipeline is complete we need a way to scrape data related to both the sequencing run and informatic pipeline output.

Illumina technology relies on a sample sheet format for describing the chemistry and biology of each discrete sample loaded onto a machine (multiplexed) in a specific sequencing run [1]. The sample sheet is always located in the root of the run folder and is named using the experiment identifier and the word `SampleSheet.csv`. It is this file that we will need in order to locate, scrape, parse, and load run specific information into a database for organized local storage.

The stack of technology was chosen to facilitate flexibility in future modifications to data organization, schema, and final representation.

First `R` was chosen to both scrape and parse the sample sheets from `AWS`. Initially only the Python library `boto3`, a SDK for `AWS S3`, seemed mature enough to aid in locating and streaming the sample sheet file objects from `AWS S3`. After some investigation, the `R` package `aws.s3` from the `cloudyr` project seemed to mirror all necessary functions and API methods [2].

The single biggest obstacle to dynamic location of the sample sheets is that `AWS S3` does not store file objects in a traditional file system. Instead, all files are found in a `bucket` and their hierarchy is encoded as a field of metadata. This immediately posed an issue as each `S3` stored sequencing run *folder* contained millions of files, many raw sequencing output. Iterating through each file handle in a single run folder would take dozens of minutes.

To solve this problem a prefix and delimiter filter combination was provided to `aws.s3::get_bucket()` which significantly limited how many objects were returned and subsequently searched through. Once the sample sheet is located it is then streamed locally and parsed with `R`.

The SQLite relational database was chosen for the data store simply because of its model of data (relational) and its comittment to efficiency, simplicity, and economy. Ultimately, the lightweight task of storing a few dozen fields for a couple hundred records needs no complicated data store [3]. `R` has a fantastic wrapper around common SQL syntax in the package `RSQLite` [4].

For this assignment a function to create the database with all tables, if it did not already exist, was created. This function will overwrite the database if the `overwrite` parameter is set to `True`. For development, the ability to start with a fresh database made testing new code very simple.

An `R` function was written to take a parsed sample sheet (in a logical data structure) and process it to fit the relational model. This function also inserts the record(s) into the database if they do not already exist. At this point in the project showing data retrieval is the only remaining task. There are examples of SQL queries at the bottom of this report.

Finally, a user-interface written in `Python` and web languages was built to query this database and render the data from a specified chosen sample sheet in the browser. This was our ultimate goal for the project as storing and retrieving data is not useful to anyone unless the results are visualized.

Setting up the Environment

After all required packages are loaded into the environment credentials must be established to access `AWS S3` since our data is hosted privately. The code was written in such a way that if the user has setup the `AWS S3 CLI` on their personal computer then they do not need to provide personal passwords as they are loaded into the environment automatically.

```
library(aws.s3)
library(itertools)
library(RSQLite)
library(stringr)

# Code to install cloudyr (not on CRAN) on a Windows OS.
#
# install.packages("aws.s3",
#                  repos=c("cloudyr" = "http://cloudyr.github.io/drat"),
#                  INSTALL_opts="--no-multiarch")

# AWS CLI must be installed and properly configured. See AWS S3 CLI manual.

# Signature must be used for steaming file objects from S3
aws.signature::use_credentials()

# Credentials must be explicitly sent to `get_bucket` methods.
credentials <- aws.signature::read_credentials()
```

Scraping AWS S3 using R

After the AWS S3 CLI has been setup on a local machine we are able to query the private buckets hosted on S3. Illumina run folders are consistently organized which makes scraping these directories rather simple. One immediate design obstacle was that in each run folder there can exist up to half a million files depending on the run output. This is due to the fact that Illuimna machines output, in parallel, thousands of small image files which are later used to *call* nucleotide bases. This function would take minutes to hours to complete as it attempted to scan a run folder for a single file.

To design around this speed obstacle we tune our GET request to AWS S3 to look for a prefix and delimiter which narrows our file search to directory level where the sample sheet exists. The sample sheet is found using a regex pattern, parsed, and returned.

```
get.sample.sheet.config <- function(run, max=1000) {
  # Null if run does not exist or connection cannot be established.
  config <- NULL

  # To minimize GET requests to AWS we must specify the prefix of the run
  # folder and the delimiter to force `get_bucket` to return only file objects
  # one directory deep. Without this, `get_bucket` will return over a million
  # object keys!
  bucket <- get_bucket(
    bucket='twinstrand-run-folders',
    prefix=paste(run, '/', sep=''),
    delimiter='/',
    max=max,
    key=credentials$default$AWS_ACCESS_KEY_ID,
    secret=credentials$default$AWS_SECRET_ACCESS_KEY
  )

  for ( object in bucket ) {
    # Skip any object that is not a SampleSheet, case-insensitive.
    if ( grepl('samplesheet', str_to_lower(object$Key)) == F ) { next }

    cat('\n[   INFO   ] Discovered', object$Key)
    cat('\n[   INFO   ] Parsing', run)
    break
  }
  # Stream the SampleSheet and parse it into an organized data structure.
  return(parse.sample.sheet.object(object))
}
```

Parsing the Sample Sheet

Next is a far more complicated task. The sample sheet is irregularly formatted. It is comma separated although does not conform to a simple data shape like a spread sheet or `data.frame`. The overall architecture of the document follows a convention used in `.ini` files but deviates in the sections for `Data` and `Reads`. Therefore we are not able to use a `data.frame`, `csv`, or `config` parser but have to write a custom function for this task.

First the `AWS S3` object is streamed locally, converted to character from binary, modeled as a `textConnection` and then split by newlines. Finally, each row/line in the resulting object is split on commas and all whitespace is trimmed from all fields.

Second, an iterable tool is used to advance through the lines of the sample sheet with the ability to advance to the next line within the loop's current instruction.

The data is parsed into appropriate data structures and finally returned in a named list.

```
parse.sample.sheet.object <- function(object) {
  # Stream the contents of the object, decompress, make text connection,
  # and read in lines. Split them on commas, trimws, and unlist.
  sample.sheet <- readLines(textConnection(rawToChar(get_object(object))))
  sample.sheet <- lapply(
    sample.sheet, function(x) trimws(unlist(strsplit(x, ','))))

  # Initalize all containers for this SampleSheet.
  config <- list(header = list(), reads = c(), settings = list(), data = list()
  )

  # Create a special iterator which can be advanced manually and exhausted.
  handle <- itertools::ihasNext(sample.sheet)
  while ( hasNext(handle) ) {
    # Split line on comma.
    line <- nextElem(handle)

    # If this is an empty line, skip!
    if ( all(line == '') ) { next }

    # If this row matches a section pattern save section name and advance.
    section_match = str_match(string=line[1], pattern='\\[(.*)\\]')
    if ( any(!is.na(section_match)) ) {
      # Enforce that section names are always lowercase going forward.
      section = str_to_lower(section_match[2])
      next
    }

    # Handle data separately for each section group. [Header] and [Settings]
    # just happen to follow .ini convention. [Reads] and [Data] do not.
    if( section %in% c('header', 'settings') ) {
      config[[section]][[gsub(' ', '', line[1])]] <- line[2]
    }
    # [Reads] are a vertical list of read lengths, at most two rows.
    else if ( section %in% c('reads') ) {
      config[[section]] <- c(config[[section]], as.numeric(line[1]))
    }
    # [Data] are represented as a subsheet with a header row and samples
    # following that header row. If we encounter our first row, save it as a
```

```
    # header, then advance to samples and save each in a zipped list with the
    # header row as names.
    else if ( section %in% c('data') ) {
      if ( length(config[[section]]) == 0 ) {
        sample_header = line
        line = nextElem(handle)
      }
      # Append a new list of sample metadata to the growing list of samples.
      config[[section]] <- append(
        config[[section]],
        list(setNames(as.list(line), sample_header)))
    }
  }
  return(config)
}
```

Creating the SQLite Database

To faciliate early development a small routine was designed to construct a third normal form relational database. The data was logically broken up into two tables, one for sequencing `runs` and another for `samples`. Sequencing runs are unique objects in time and space that cannot be represented twice. Within a sequencing run are samples. Samples can be replicated both within a run and on different runs. Our primary key for both tables will be the `run` identifier and allow us to `JOIN` both table to make queries related to the sequencing runs and the samples that were included.

First, a database is created at the path specified in the `name` argument if it does not yet exist. The two tables for this project are checked and if the user has asked for them to be dropped, they are.

Finally both tables are created using `SQL` syntax to match the data structure parsed from the sample sheets.

```
make.sample.sheet.database <- function(name, overwrite=F) {
  cat('\n[  CONNECT ] Connecting to db:', name)
  conn <- dbConnect(RSQLite::SQLite(), dbname=name)

  for ( table in c('runs', 'samples') ) {
    if ( table %in% dbListTables(conn) && overwrite == T ) {
      cat('\n[  WARNING ] Dropping table:', table)
      dbRemoveTable(conn, table)
    }
  }

  if ( !'runs' %in% dbListTables(conn) ) {
    cat('\n[   INFO   ] Creating table:', 'runs')
    dbGetQuery(conn=conn,
      'CREATE TABLE runs
        (run TEXT,                      IEM1FileVersion INTEGER,
         InvestigatorName TEXT,         ExperimentName TEXT,
         Date TEXT,                     Workflow TEXT,
         Application TEXT,              Assay TEXT,
         Description TEXT,              Chemistry TEXT,
         Paired BIT,                    ReadLength INTEGER
         CreateFastqForIndexReads BIT   BarcodeMismatches INTEGER
         a)')
  }

  if ( !'samples' %in% dbListTables(conn) ) {
    cat('\n[   INFO   ] Creating table:', 'samples')
    dbGetQuery(conn=conn,
      'CREATE TABLE samples
       (run TEXT,             Sample_ID TEXT,
        Sample_Name TEXT,     Library_ID TEXT,
        Sample_Project TEXT,  Description TEXT,
        index1 TEXT,          index2 TEXT,
        Target_Set TEXT,      Reference_Name TEXT,
        Read_Structure TEXT,  Fixed_Umi_Set_Name TEXT)')
  }
}
```

Instantiating Database

Calling the function `make.sample.sheet.database` creates the SQLite database and will drop any tables named `run` or `samples` if they exist and `overwrite=T`. The use of `cat` statements were used to inform the user of actions performed which aided in the design of this project and will continue to help users of this code.

```
dbname <- 'sample_sheets.sqlite'

make.sample.sheet.database(dbname, overwrite=T)
```

```
##
## [  CONNECT ] Connecting to db: sample_sheets.sqlite
## [  WARNING ] Dropping table: runs
## [  WARNING ] Dropping table: samples
## [   INFO   ] Creating table: runs
## [   INFO   ] Creating table: samples
```

```
## data frame with 0 columns and 0 rows
```

Adding a Sequencing Run to the Database

The last function defined performs the task of appending a sample sheet (as identified by its `run` identifier) to both tables in the database only if it does not already exist in the `runs` table. A small amount of preprocessing of the sample sheet data structure is needed since relational databases are not designed to store objects similar to lists. For instance, `Reads` is a list of integers either one or two long. If two integfers are present they are the same value. For storing in our SQLite database we must first store the length of the list in a bit and only report one of the values as the `ReadLength`.

All of the sample sheet metadata is added on1ce to the `runs` table and with it many records for samples to the `samples` table. All records are given the key `run` identifier which has been chosen to avoid record collisions in conformance with 3NF.

```
add.run.to.db <- function(run, dbname) {
  conn <- dbConnect(RSQLite::SQLite(), dbname=dbname)

  # Check if run record already exists in database, if so skip this iteration.
  response <- dbGetQuery(conn=conn,
    paste('SELECT EXISTS(
           SELECT * FROM runs
           WHERE run = "', run, '")',
          sep=''))

  if ( response == T ) {
    cat('\n[   INFO   ] Run already in db', run)
    return(NULL)
  }

  # Since the record does not exist we will scrape and parse the source file
  # from AWS and prepare it for inclusion into our database.
  config <- get.sample.sheet.config(run)

  # Check to see if Illumina reads are paired-end or not. Set a bit.
  if ( length(config[['reads']]) == 2 ) { paired = 1 } else { paired = 0 }

  # Compose a data.frame of all fields and their values to append to the
  # run database. This must be in the order the SQLite database expects.
  run.table <- data.frame(list(
    run=run,
    config[['header']],
    Paired=paired,
    ReadLength=config[['reads']][1]
  ))

  # Write record to database, ensuring that we append.
  dbWriteTable(conn, 'runs', run.table, append=T)
  cat('\n[ DB WRITE ] Run', run, 'added to db', dbname)

  for ( sample in config$data ) {
    # Compose a data.frame of all samples with the primary key `run`` and append
    # to the samples database.
    sample.table <- data.frame(list(run=run, sample))

    # NOTE: hack since Illumina named a field `index` which collides with
    # SQL syntax.
    colnames(sample.table)[match('index', colnames(sample.table))] <- 'index1'

    dbWriteTable(conn, 'samples', sample.table, append=T)
  }
  cat('\n[ DB WRITE ] Added', length(config$data), 'sample records to db', dbname)
}
```

Running all Code

A local register of run names must be specified for the above functions to prevent unwanted runs from automatically being included in the database. For the sake of this report, four random run names were chosen to show the robust operation of the code written above.

```
runs <- c(
  '170126_NB501910_0002_AHHJWLBGX2',
  '170210_NB501910_0004_AHJLMYAFXX',
  '170217_NB501910_0005_AHJY3LAFXX',
  '170302_NB501910_0006_AHJK23AFXX'
)

for ( run in runs ) { add.run.to.db(run, dbname) }
```

```
##
## [   INFO   ] Discovered 170126_NB501910_0002_AHHJWLBGX2/aml001-SampleSheet.csv
## [   INFO   ] Parsing 170126_NB501910_0002_AHHJWLBGX2
## [ DB WRITE ] Run 170126_NB501910_0002_AHHJWLBGX2 added to db sample_sheets.sqlite
## [ DB WRITE ] Added 1 sample records to db sample_sheets.sqlite
## [   INFO   ] Discovered 170210_NB501910_0004_AHJLMYAFXX/amg001_SampleSheet.csv
## [   INFO   ] Parsing 170210_NB501910_0004_AHJLMYAFXX
## [ DB WRITE ] Run 170210_NB501910_0004_AHJLMYAFXX added to db sample_sheets.sqlite
## [ DB WRITE ] Added 5 sample records to db sample_sheets.sqlite
## [   INFO   ] Discovered 170217_NB501910_0005_AHJY3LAFXX/dup010_011_SampleSheet.csv
## [   INFO   ] Parsing 170217_NB501910_0005_AHJY3LAFXX
## [ DB WRITE ] Run 170217_NB501910_0005_AHJY3LAFXX added to db sample_sheets.sqlite
## [ DB WRITE ] Added 14 sample records to db sample_sheets.sqlite
## [   INFO   ] Discovered 170302_NB501910_0006_AHJK23AFXX/dup012_SampleSheet.csv
## [   INFO   ] Parsing 170302_NB501910_0006_AHJK23AFXX
## [ DB WRITE ] Run 170302_NB501910_0006_AHJK23AFXX added to db sample_sheets.sqlite
## [ DB WRITE ] Added 20 sample records to db sample_sheets.sqlite
```

One feature of note is that the code will skip any run that already exists in the database. This prevents the duplication of records every time a scraping function is performed.

```
for ( run in runs ) { add.run.to.db(run, dbname) }
```

```
##
## [   INFO   ] Run already in db 170126_NB501910_0002_AHHJWLBGX2
## [   INFO   ] Run already in db 170210_NB501910_0004_AHJLMYAFXX
## [   INFO   ] Run already in db 170217_NB501910_0005_AHJY3LAFXX
## [   INFO   ] Run already in db 170302_NB501910_0006_AHJK23AFXX
```

Sample SQL Query

This example query shows data retrieval from the sample sheet database. The query joins both the `run` and `samples` tables to find all samples (only a subset of data is shown) that were loaded by operator `Lindsey`.

```
conn <- dbConnect(RSQLite::SQLite(), dbname=dbname)
query <- "
  SELECT *
  FROM samples
  INNER JOIN runs
  ON runs.run = samples.run
  WHERE InvestigatorName = 'Lindsey'
"
result <- dbGetQuery(conn, query)
result[c('Sample_Name', 'Sample_Project', 'Description', 'index1')]
```

```
##    Sample_Name Sample_Project                         Description   index1
## 1          6_6         AMG001  1:1 ratio genomic:transgene probes CGACTGGA
## 2          7_7         AMG001  1:5 ratio genomic:transgene probes GATAGACA
## 3          8_8         AMG001 1:10 ratio genomic:transgene probes GTCGTAGA
## 4          9_9         AMG001 1:20 ratio genomic:transgene probes TGGAACAA
## 5        10_10         AMG001 1:40 ratio genomic:transgene probes AATGTTGC
## 6        11_11         dup010                           SOP oligos ATCATTCC
## 7        20_20         dup011                          350 covaris GTCTGTCA
## 8        21_21         dup011                        Loeb shearing TGGCTTCA
## 9        22_22         dup011                          250 covaris ACACGACC
## 10       23_23         dup011                          300 covaris ATTGGCTC
## 11       24_24         dup011                          350 covaris CGGATTGC
## 12       12_12         dup010                           SOP oligos CGACACAC
## 13       13_13         dup010                     adapter control AAACATCG
## 14       14_14         dup010                     adapter control ACAAGCTA
## 15         3_3         dup010                index primer control AAGGTACA
## 16         4_4         dup010                index primer control AGATCGCA
## 17         1_1         dup011                          250 covaris AACGTGAT
## 18         2_2         dup011                        Loeb shearing CGCTGATC
## 19       19_19         dup011                          300 covaris GCCACATA
```

Future Tasks and Goals

All that remains to be done is for this database to be used for programmatic access and integration into the companies stack of other analysis tools. As investigators craft hypothesis an SQL query can be constructed to answer the question. If the question must be asked often then a routine can be written to do this efficiently and even embedded on the web app so others in the company can utilize the database.

Ultimately, more tables will be needed to store other information related to each sequencing run as there is plenty more metadata to collect, store, and retrieve related to each sequencing run output.

Bibliography

1. "Generating the Sample Sheet." Generating the Sample Sheet. Illuimna, n.d. Web. 12 Apr. 2017. url: https://support.illumina.com/help/SequencingAnalysisWorkflow/Content/Vault/Informatics/Sequencing_Analysis/CASAVA/swSEQ_mCA_GeneratingSampleSheet.htm
2. "Amazon Simple Storage Service (S3) API Client." Cloudyr, aws.s3, n.d. Web. 12 Apr. 2017 url:https://github.com/cloudyr/aws.s3
3. "Appropriate Uses for SQLite." SQLite, n.d. Web. 12 April. 2017. url:https://sqlite.org/whentouse.html
4. "R interface for SQLite https://rstats-db.github.io/RSQLite." RSQLite, n.d. Web. 12 Aprl. 2017. url:https://github.com/rstats-db/RSQLite