# A study of access modes for cloud-based data

# Introduction: goals

Substantial parts of the SRA have been moved from local POSIX filesystem into cloud-based object stores. The SRA toolkit has been developed with the POISX model in mind and appeared to have degraded performance on object stored the cloud. This study has been made to test several hypotheses regarding possible improvements in the performance of the SRA toolkit. The areas of interest are the storage size, the speed and stability of data transmission, and their variation based on the layout of the objects.

## Smaller footprint

As originally developed in 2000s, the SRA uses gzip as the method of compressing column blobs. Since then, additional and possibly better methods became available and the hardware became more powerful offsetting the performance degradation of the loaders caused by using more advanced algorithms. We experimented with alternative compression methods to find out potential benefits.

## Lower error rate, improved stability, faster downloads

Based on the stream of support request from users of the SRA toolkit, the transition of the SRA data to cloud has caused an increase in network timeouts and decrease in the download speeds. We performed an analysis of client-side logs created while downloading some SRA runs by fastq-dump to identify the sources of slowdown. We also tested different server-side layouts of the SRA-like objects to assess their influence on the speed and reliability of downloads.

# Footprint

[VDB-4424](https://jira.ncbi.nlm.nih.gov/browse/VDB-4424) provides results of applying multiple modern compression algorithms to a set of ASCII reads extracted from an SRA run. The research shows that VDB’s current compression ratio has room for improvement. Zstd seems to promise the biggest gains.

We took an approach that is based on using bzip, the compression method alternative to gzip and currently supported by the VDB schema.

Wolfgang:

I wrote a tool to test how much compression can be gained by switching from gzip-encoding to bzip-encoding in the schema.

The tool is located on the engineering branch of ncbi-vdb under py-vdb/

It consists of these files : L11-table\_copy.py, record\_sizes.py, tbl\_copy\_1.vschema, tbl\_copy\_2.vschema and run.sh.  
'run.sh' takes one argument: an accession  
It 'prefetches' the accession into the current directory.  
It makes 2 copies of the accession one with gzip-encoding (tbl\_copy\_1.vschema) and one with bzip-encoding (tbl\_copy\_2.vschema). *Note that it only copies the columns necessary to produce FASTQ and stores them in the ASCII format.*  
It records the following information in a sqlite-database (created if it does not exist):  
Accession, original size, gzip-encoded size, bzip-encoded size, percentage of bzip-encoded vs. original size  
At the end run.sh deletes all prefetched/created files.

To test multiple accession, I created a simple text file with one accession per line (for instance acc.txt).  
Then fed the list via xargs to the run.sh-script: 'xargs -L1 -a acc.txt ./run.sh'  
The script will create a sqlite-database named 'data.db' in the current directory.

This small sqlite-database is attached to the ticket: <https://jira.ncbi.nlm.nih.gov/secure/attachment/435924/data.db>

To query it run 'sqlite3 --column --header data.db "select \* from data;"'  
it looks like this:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **acc** | **org\_size** | **copy1** | **copy2** | **copy2/org\_size, %** | **copy2/copy1, %** |
| SRR13340286 | 23,177,653 | 24,577,025 | 7,005,209 | 30 | 29 |
| SRR13340289 | 21,846,855 | 23,146,271 | 5,571,588 | 25 | **24 <- max benefit** |
| SRR13340310 | 23,592,792 | 25,073,085 | 7,161,467 | 30 | 29 |
| SRR13340412 | 21,571,113 | 22,849,861 | 6,036,750 | 27 | 26 |
| SRR13346827 | 22,679,937 | 23,975,869 | 7,160,206 | 31 | 30 |
| SRR13346830 | 21,709,347 | 23,036,549 | 5,992,158 | 27 | 26 |
| SRR13347107 | 21,743,902 | 22,987,343 | 8,550,125 | 39 | 37 |
| SRR13347129 | 24,008,760 | 25,410,371 | 9,447,410 | 39 | 37 |
| SRR13423495 | 21,062,689 | 19,495,189 | 5,738,314 | 27 | 29 |
| SRR13423498 | 19,351,186 | 17,963,055 | 4,825,136 | 24 | 27 |
| SRR13510113 | 8,215,060 | 7,773,682 | 7,014,990 | 85 | 90 |
| SRR13526958 | 30,564,408 | 31,839,488 | 15,030,482 | 49 | 47 |
| SRR13527118 | 47,345,894 | 49,044,777 | 26,705,873 | 56 | 54 |
| SRR13527320 | 33,638,238 | 34,727,165 | 17,876,100 | 53 | 51 |
| SRR13527415 | 45,303,045 | 47,197,363 | 21,165,739 | 46 | 45 |
| SRR13527519 | 47,372,178 | 49,418,899 | 23,408,584 | 49 | 47 |
| SRR13533219 | 15,719,347 | 7,331,013 | 4,324,280 | 27 | 59 |
| SRR13533372 | 18,133,955 | 8,412,906 | 7,643,472 | 42 | 91 |
| SRR13534253 | 17,054,225 | 7,742,147 | 4,202,147 | 24 | 54 |
| SRR13689366 | 51,599,852 | 53,514,096 | 27,807,867 | 53 | 52 |
| SRR13510047 | 19,979,047 | 18,940,432 | 17,157,327 | 85 | 91 |
| SRR13533182 | 21,538,945 | 9,795,557 | 5,778,562 | 26 | 59 |
| SRR13533900 | 20,307,772 | 9,485,333 | 7,818,739 | 38 | 82 |
| SRR13533969 | 25,035,444 | 11,693,133 | 8,109,292 | 32 | 69 |
| SRR13534892 | 28,399,307 | 14,358,840 | 13,199,822 | 46 | 92 |
| SRR13689757 | 59,097,178 | 61,619,976 | 29,737,131 | 50 | 48 |
| SRR13527480 | 88,847,370 | 91,468,683 | 55,432,873 | 62 | 61 |
| SRR13533701 | 24,925,503 | 12,362,478 | 11,130,782 | 44 | 90 |
| SRR13533967 | 26,441,134 | 12,450,278 | 11,069,130 | 41 | 89 |
| SRR13534046 | 24,594,924 | 11,720,365 | 10,356,636 | 42 | 88 |
| SRR13360583 | 69,414,156 | 70,067,808 | 66,698,251 | 96 | 95 |
| SRR13360599 | 70,178,074 | 70,852,096 | 67,599,695 | 96 | 95 |
| SRR13360658 | 68,661,601 | 69,277,396 | 66,054,146 | 96 | 95 |
| SRR13360652 | 71,383,503 | 72,082,254 | 68,903,676 | 96 | 96 |
| SRR13360578 | 69,491,085 | 70,177,430 | 66,954,603 | 96 | 95 |
| SRR13360631 | 69,073,949 | 69,719,662 | 66,441,388 | 96 | 95 |
| SRR13360645 | 67,218,142 | 67,841,735 | 64,699,077 | 96 | 95 |
| SRR13360661 | 63,424,657 | 64,011,968 | 60,946,545 | 96 | 95 |
| SRR13360582 | 65,667,558 | 66,267,486 | 63,435,130 | 96 | 96 |
| SRR13360574 | 70,324,867 | 71,006,444 | 67,692,867 | 96 | 95 |
| SRR13350100 | 95,761,694 | 95,439,038 | 84,216,495 | 87 | 88 |
| SRR13360587 | 124,817,330 | 126,422,015 | 120,768,100 | 96 | 96 |
| SRR13360683 | 176,812,317 | 177,612,572 | 169,665,766 | 95 | 96 |
| SRR13344572 | 359,230,103 | 364,272,594 | 287,404,988 | 80 | 79 |
| SRR13360634 | 121,070,416 | 122,570,963 | 117,089,911 | 96 | 96 |
| SRR13360635 | 149,284,004 | 151,288,426 | 144,050,290 | 96 | 95 |
| SRR13441258 | 192,228,859 | 155,408,588 | 133,484,285 | 69 | 86 |
| SRR13532992 | 79,971,384 | 38,686,682 | 35,425,853 | 44 | 92 |
| SRR13622365 | 20,520,123 | 8,745,831 | 4,985,159 | **24 🡨 max benefit** | 57 |
| SRR13689481 | 233,115,064 | 240,213,298 | 120,338,777 | 51 | 50 |
| SRR2339625 | 2,045,965,011 | 1,910,178,264 | 1,871,933,008 | 91 | 98 |
| SRR2339676 | 1,538,311,868 | 1,377,147,708 | 1,202,777,965 | 78 | 87 |
| SRR1979900 | 1,148,453,419 | 1,179,651,337 | 988,686,474 | 86 | 84 |
| SRR1734369 | 3,673,281,498 | 3,505,304,361 | 2,274,389,922 | 61 | 65 |
| SRR1653598 | 1,394,969,490 | 1,267,218,855 | 1,078,299,167 | 77 | 85 |
| SRR7814422 | 349,383,720 | 356,374,943 | 307,420,081 | 87 | 86 |
| SRR7814404 | 361,011,966 | 368,808,824 | 334,821,707 | 92 | 91 |
| SRR7814416 | 383,771,863 | 391,805,750 | 347,769,917 | 90 | 89 |
| SRR7814412 | 392,089,171 | 401,455,897 | 359,590,971 | 91 | 90 |
| SRR7814423 | 365,161,094 | 372,753,951 | 337,325,294 | 92 | 90 |

According to the table above, the compression gain highly depends on the data. The result can be as small as 24% of the original or as big as 96%.

The averages of this selection of runs are 63% for copy2 / org\_size and 72% for copy2 / copy1. This is what we can expect to achieve by simply changing the compression from gzip to bzip in the current SRA schema. In order to support other compression algorithms mentioned in VDB-4424, the VDB schema language and the run-time code would have to be changed.

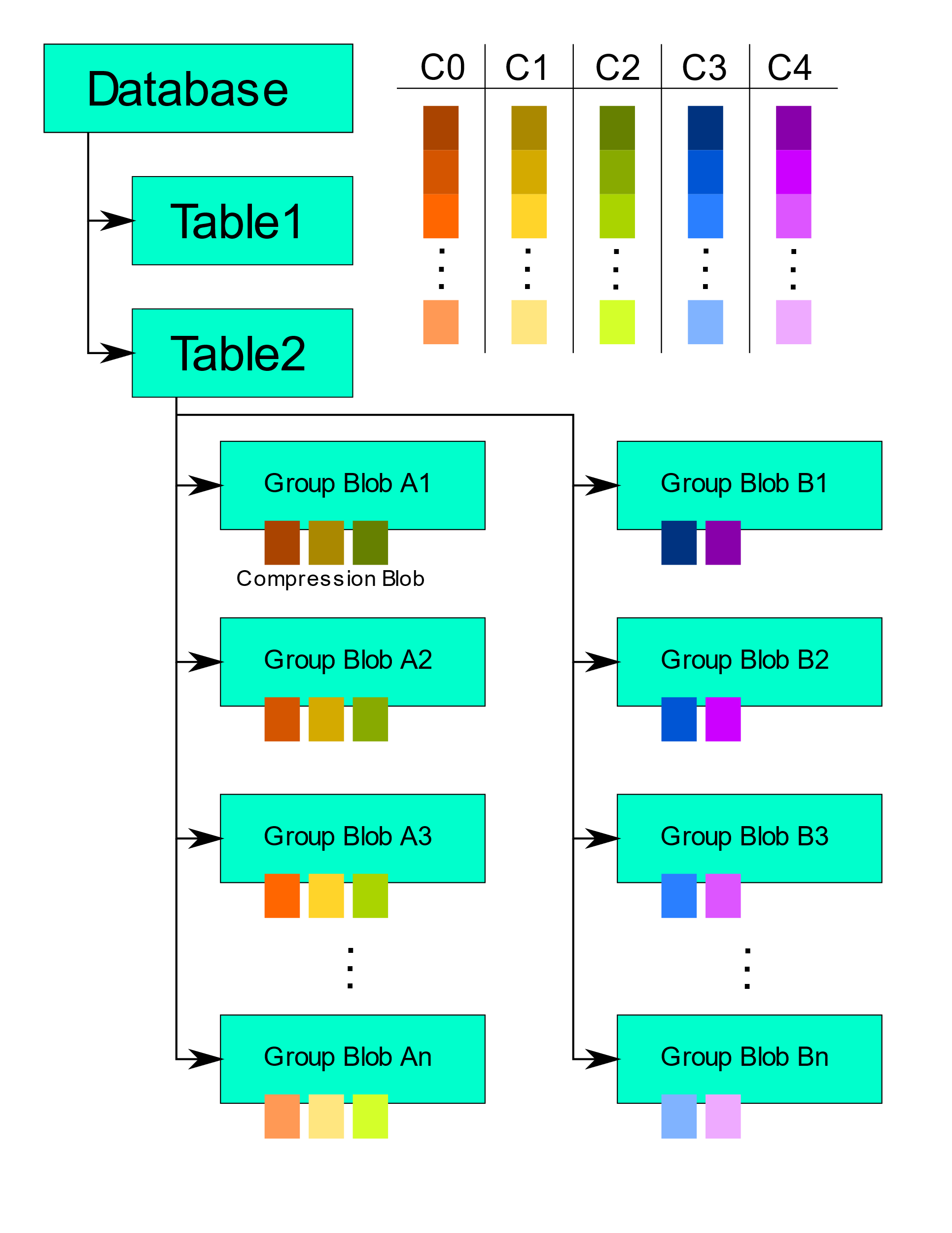
One further area of the research would be to experiment with different compression level settings of the algorithms. Setting the level of compression can also be supported in the schema language.

# Single file with partial reads vs multiple files with full reads

One of the hypotheses this study was supposed to test is that when reading an object from a cloud-based storage it is more efficient and/or reliable to represent the object as a set of smaller files. The reasoning is that using HTTP requests to download complete smaller files is better than an equivalent set or partial requests on a single file.

To test the hypothesis, we created a script (writer2.py) that uses the Python binding of the VDB library to read an unaligned SRA run and produces a set of files representing a partitioning of the SEQUENCE table into *blob groups*.

A blob group is a slice of a set of columns (a matrix), as demonstrated on the following picture.



Our script allowed us to define different groupings of columns based on an internal hard-coded schema.

Note that the compression was applied to each column in a particular blob group, and to the blob group itself.

# Error rates

We created a set of scripts that perform fastq-dump on a selection of SRA runs read from multiple source locations: AWS and 2 different locations on Cloudian. The goal was to measure networking error rates (timeouts on connect or read) in relation to the source of the data.

We selected 10 accessions of about the same row count (about 300K rows) loaded at around the same time (early 2020), all unaligned (single SEQUENCE table).

We ran 4 different scripts:

1. Fastq-dump with the default URL from SDL.

This gave us a mix of ~80% AWS, ~20% Cloudian.

1. Fastq-dump with a URL as returned by SDL forced to point to Cloudian.
2. Fastq-dump with a URL into the study-specific location on Cloudian.

We requested a region of 50G on NCBI Cloudian and uploaded copies of the accessions to it. We uploaded a copy of every original accession to a bucket in the region and gave the corresponding URL to fastq-dump.

1. A Python script doing fastq-dump on a set of group blobs located in Cloudian.

This script, reader2.py, performed the equivalent of fastq-dump on sets of blob group files created by the writer2.py script. Each set was created to represent a single accession. The schema used by writer2.py combined READ, QUALITY and NAME columns in a single column group.

The reader2.py script downloaded the group blob files one by one using full-file HTTP GET requests though standard Python HTTP client.

As opposed to the scripts that use the VDB library to connect and read, this script made no attempt to recover from any errors.

The scripts were run in a loop separated by a 20 second pause, for approximately 4 days. We recorded execution time and exit code in an SQLite database.

The scripts were run on computers outside of NCBI network.

The following numbers, broken down by the script, represent the counts of failed runs as percentage of total runs in the 4-day period.

|  |  |  |  |
| --- | --- | --- | --- |
| Script | Total runs | Failed (\*) runs | % |
| A | 30,110 | 1 | 0 |
| B | 25,167 | 94 | 0.3 |
| C | 27,723 | 29 | 0.1 |
| D | 34,326 | 70 | 0.2 |

(\*) By “failed” we mean a non-0 exit code or execution time longer than 100 seconds; the average run time across all scripts was about 8.8 seconds.

Note that whenever the Python script errored out, the reason stated by the library was always “Connection unexpectedly closed by the remote endpoint”. The scripts A through C use VDB which attempts to recover from such situations, so when they fail the reported reason is usually a VDB timeout.

The database with the raw data collected for this experiment is located in /panfs/traces01/sra\_review/scratch/raetzw/script\_dbs/stab2.db

It has a single table “log” with columns “reader”, “source”, “ret\_code”, “runtime”, etc. Each row represents a single execution of a single script. The scripts are identified by the combination of values in columns “reader” and “source” as per the following table:

|  |  |  |
| --- | --- | --- |
| Script | reader | source |
| A | “fastq” | “sdl” |
| B | “fastq” | “cloudian” |
| C | “fastq\_spec” | “cloudian” |
| D | “vdb3” | “cloudian” |

# Speed of downloads

In this part of the study, we measured the time it takes to execute fasqt-dump, with variations in the source of data, as well as the layout, similarly to the previous section. Some of the data were collected in the same 4-day run as the error rate experiment.

The questions to answer were:

* How does the runtime depend on the type of cloud (AWS vs Cloudian)?
* Can we gain speed by storing the objects as sets of group blob files?
* Can we gain speed further by downloading group blobs in parallel?
* In case of a single file, how does runtime differ between sequential and random order of partial reads?

|  |  |  |  |
| --- | --- | --- | --- |
| Script | avg(runtime), sec | max(runtime), sec | min(runtime), sec |
| A | 8.2 | 123.6 | 3.6 |
| B | 9.7 | 352.5 | 3.6 |
| C | 11.1 | 2,392.8 | 3.5 |
| D | 7.1 | 453.7 | 2.5 |

An unoptimized Python script reading a collection of group blobs beats an optimized C library reading from a single file, in download speed, from the same bucket (D vs C).

To measure how download times depend on sequential/random order of partials reads from a single file, we ran a separate experiment. It involved running 5 scripts:

* Download all group blob files representing an accession.
* Make a sequence of partial downloads on a single file located on Cloudian, in sequential order
* Make a sequence of partial downloads on a single file located on Cloudian, in random order
* Make a sequence of partial downloads on a single file located on AWS, in sequential order
* Make a sequence of partial downloads on a single file located on AWS, in random order

# Conclusions

We can expect a 30% reduction in the storage footprint of SRA object by upgrading compression used by VDB. This would require a minor change in the schema language. An additional improvement may be to support varying level of compression in the schema.

The error rate is correlated with the source being AWS or Cloudian. Changing representation of the data to support full requests versus partial requests does not seem to affect the error rate.