A study of access modes for cloud-based data

Anatoly Boshkin, Wolfgang Raetz

5/04/2021

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 POSIX model in mind and appeared to have degraded performance on objects stored the cloud. This study serves 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 investigated 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 the cloud has caused an increase in network timeouts and decrease in the download speeds. We performed experiments downloading some SRA runs from different cloud-sources. We also tested different server-side layouts of the SRA-like objects to assess their influence on the speed and reliability of downloads.

Notes

All experiments were run from outside of the NCBI network, using Linux on non-virtual machines.

There is a large variance in the download-speeds in performing the same request multiple times. The variance is much higher when downloading from Cloudian than from AWS. We are not able to pinpoint the infrastructure component responsible for that.

Footprint

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 best compromise between speed and compression.

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 I 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`
- 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.

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

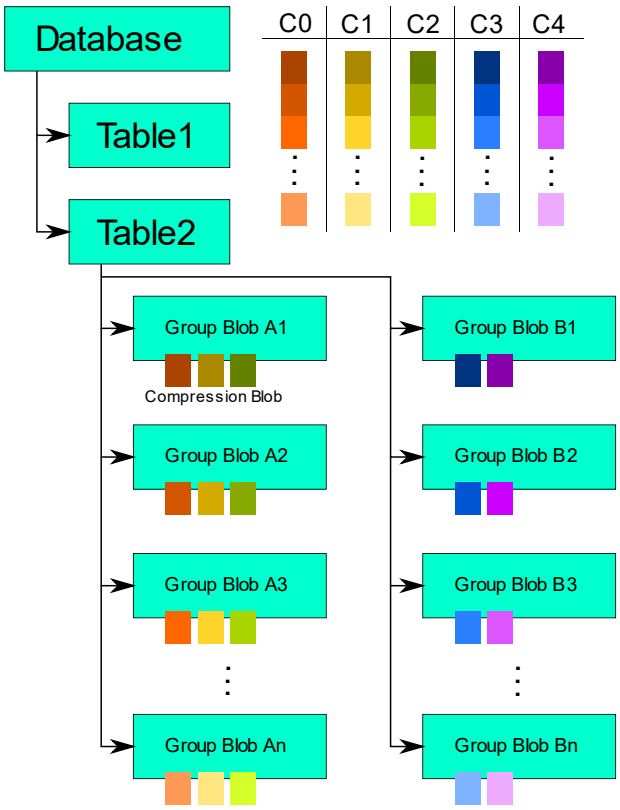
Single file with partial reads vs multiple files with full reads

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

Single files with partial reads is the mode of operation of the current SRA tools. Multiple files read in one request each is what we are simulating with a set of Python scripts.

To test the hypothesis, we created a Python script [writer2.py] that uses the Python binding of the VDB library to read an SRA run (single or multiple table) and produces a set of files representing a partitioning of the data in each table into *group blobs*.

A group blob 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 group blob, and to the serialized group blobs themselves.

To accompany writer2.py, we created another script, reader2.py which produces FASTQ output from the group-blob representation. This is equivalent to running the fastq-dump tool with the “--split-spot“ option. We matched the outputs of reader2.py with outputs of fastq-dump to verify the correctness of the group-blob representation.

Error rates

We created a shell script stability_db.sh[1] that invokes either fastq-dump or reader2.py 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, each one a flat table SRA archive (not CSRA database).

We ran the script stability_db.sh with 4 different arguments:

A. 'stablility_db.sh fastq_sdl' invokes fastq-dump with the default URL from SDL.
   This gave us a mix of ~80% AWS, ~20% Cloudian.
B. 'stability_db.sh fastq' invokes fastq-dump with a URL as returned by SDL forced to point to Cloudian.
C. 'stability_db.sh fastq_special' invokes fastq-dump with a URL into the study-specific location on Cloudian. For this location 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 used the corresponding URL.
D. 'stability_db.sh fastq_vdb3' invokes a Python script on a set of group blobs located in Cloudian. This Python script, reader2.py, performs the equivalent of fastq-dump on sets of group blob 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 fastq-dump, this Python script makes no attempt to recover from any errors.

The script 'stability_db.sh' internally runs an endless loop calling the selected variation separated by a 20 second pause. We ran each instance of the script in a screen-session for approximately 4 days. We recorded execution time and exit code in a common SQLite database.

All fastq-dump instances used in this experiment were debug-builds with a deep level of logging turned on.

The following numbers, broken down by arguments A…D, represent the counts of failed downloads as percentage of total attempts in the 4-day period. 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.

Script	Total attempts	Failed attempts	%
A	30,110	1	0
B	25,167	94	0.3
C	27,723	29	0.1
D	34,326	70	0.2

[1] The "db" in the name of the script indicates that it stores the results in a database (SQLite). It does not mean that we are processing SRA databases.

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 fastq-dump which attempts to recover from such situations, so when it fails the reported reason is usually a VDB library 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"

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.

Speed of downloads

In this part of the study, we measured the time it takes to execute 'fastq-dump' and 'reader2.py,' with variations in the source of data, as well as the layout. 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)?
- In case of partial reads from a single file, how does the runtime differ between sequential and random order?
- 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?

The fastq-dump tool used was the release build, with client-size caching turned on (which is the default).

We performed 2 different experiments, one involving single-table SRA archives located in AWS and on Cloudian, and another involving CSRA archives on Cloudian. We did not do AWS in the second case.

fastq-dump vs. reader2.py, single-table SRA, Cloudian vs. AWS

Here are the results of analyzing the data produced by the scripts A…D for 4 days:

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

The outlier in line C is likely to do with a network timeout and a successful recovery from it, which may feature multiple attempts to reconnect, with increasing pauses and timeout values.

While the average runtimes fall into the same ballpark, it is important to note that that A, B and C used fastq-dump which performs client-side caching inside the VDB library, whereas D (a Python script) did not need any caching. With the client-side caching turned off the run time of fastq-dump on the same accessions was about 30% longer. The effect on CSRA databases, however, was much more dramatic, as we will mention in the next section.

We assume B, C and D (all Cloudian via Nginx) employ some form of server-side caching; we are not sure about the A (mostly AWS). A detailed look at the recorded execution times shows the apparent effect of server-side caching.

The effect of server-side caching is hard to evaluate based on our limited knowledge of the corresponding configurations on the SRA side but a typical case that we have observed multiple times shows that fastq-dump run on a "fresh" accession takes about 10-15% longer than that other runs on the same archive done in quick succession.

The purpose of this experiment was to find out if avoiding partial access leads to a higher speed of conversion into FASTQ. We did not have the time to develop an optimized tool to match fastq-dump and

the VDB library; we wrote our experimental client tool in Python and did not use any parallel processing. Even with these disadvantages, the Python script (D) shows higher speeds downloading from Cloudian.

The processing speed is correlated with the source being AWS or Cloudian. On average fastq-dump conversions from AWS are about 20% faster and have less variance. Avoiding partial downloads improves the overall processing speed, even though the tool is written in Python and does not use parallel processing.

fastq-dump vs. reader2.py, CSRA on Cloudian

A major difference between accessing single-table SRA and CSRA databases is that in order to produce a FASTQ output the client has to perform multiple join operations between sequence, alignment, and reference tables, including joins with external reference databases. This creates very different access patterns at the VDB blob level. Client-side caching had to be introduced to the VDB library to address the negative effects of these join operations.

To model the new approach with our Python tool, we created a new blobbing scheme to avoid join operations. We ran writer2.py against two CSRA archives to create blob-group representations of their SEQUENCE, PRIMARY_ALIGNMENT and REFERENCE tables. Then, we ran the stability_db scripts in tw modes corresponding to C and D introduced in the previous sections. "C" ran fastq-dump against a CSRA archive located in our team's bucket Cloudian. "D" ran reader2.py against the group-blob representation uploaded to the same bucket.

Since the point of the study is to compare access times and recreating fastq from CSRA in a Python script was not practical, we focused on only the time the tools spend in actual downloads.

To achieve that, we instrumented fastq-dump (or more precise, the VDB library it uses) to measure time lapsed from the moment an HTTP GET request is sent to the moment the corresponding data is completely read from the response. At the end of an execution of fastq-dump, it reported the total time spent downloading.

The Python script did not process the received data in any way and collected the download time similarly.

NOTE It is very important that fastq-dump used client-side caching. With caching turned off, a single run of fastq-dump ran for more than 2 hours before we shut it down. Remember that client-side caching carries a cost in terms of disk space, which often comes as a surprise for the users, especially when they try do run multiple downloads in parallel.

The scripts ran overnight and produced these results:

Script	avg(runtime), sec	avg(dnldtime), sec
C	14.64	6.77
D	12.32	11.83

The difference of data volume of each download between C and D was within 5%. This may be specific to the particular accessions that we chose here and not represent a general case. We chose these accessions based on their byte size in order to stay within limitations on the size of the data in our Cloudian bucket.

When using VDB to access CSRA, client-side caching seems vital, especially for bigger archives, and does cost disk space. We were able to achieve the same download time as VDB (with caching), in our Python client using group-blob files (without any caching).

Random vs. sequential using partial access vs. non-partial access

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 instances of the timing_db.sh - scripts:

The script can be found here: timing_db.sh

E. 'timing_db.sh not_partial': Download all group blob files representing a single-table SRA accession from Cloudian (similar to D, but with different grouping of the columns) aka non-partial download
F. 'timing_db.sh partial_cloudian_seq': Make a sequence of partial downloads from a single file located on Cloudian, in sequential order
G. 'timing_db.sh partial_cloudian_rand': Make a sequence of partial downloads from a single file located on Cloudian, in random order
H. 'timing_db.sh partial_amazon_seq': Make a sequence of partial downloads from a single file located on AWS, in sequential order
I. 'timing_db.sh partial_amazon_rand': Make a sequence of partial downloads from a single file located on AWS, in random order

The results have been recorded to a Sqlite-database: timing.db

Script	avg(runtime), sec	max(runtime), sec	min(runtime), sec
E	197.3	2024.7	33.9
F	191.9	3234.2	53.8
G	206.0	4373.9	60.3
H	80.7	179.2	39.4
I	82.0	276.8	35.9

Access to AWS is always faster than access to Cloudian, no matter what access-method has been used. There is no clear winner between partial vs. non-partial downloads, or different order of partial downloads (sequential vs. random). Even if outliers are removed from the average, the picture does not change.

We did not observe significant differences in the speed of partial downloads from a single file compared to full-file downloads of the same data represented as a set of group-blob files.
The download speed is correlated with the source being AWS or Cloudian. On average downloads from AWS are about 2.5 times faster.
Partial downloads in sequential order (increasing file offset) do not differ in speed from downloading the same data-ranges in random order. Server-side caching is one possible explanation.

Sequential vs. parallel download of group-blobs

This experiment measures how much we can gain by using parallel downloads of group-blob-files.

We were running the same script stability_db.sh with 2 different arguments

J. 'stability_db.sh fastq_vdb3': to download sequentially (same as D. from above)
K. 'stability_db.sh fastq_vdb3_par': to download columns in parallel. For this we created a new set of group-blobs. We placed every column into a separate column-group. Each column can now be downloaded in a separate thread. To enable that, we added threads to the reader2.py script. This is off by default and can be turned it on via a command-line-parameter (-p).

We run 2 instances of stability_db.sh over 1 night. The results have been recorded in this Sqlite-file:

/panfs/traces01/sra_review/scratch/raetzw/script_dbs/stab3.db

All runs were single-table SRA accessions

Script	avg(runtime), sec	max(runtime), sec	min(runtime), sec
J	6.7	607.1	2.4
K	6.5	1509.3	2.5

The parallel download is only slightly faster than sequential download. Python has a global interpreter lock, which might interfere with the results. A better experiment would be to code reader.py in C++. There might be a download-rate-limiter per IP-address in our infrastructure causing this result.

Parallelizing downloads of group-blob-files does not seem to improve the overall time significantly. This may be due to using multi-threading in Python, or a rate-limiter by IP-address in the infrastructure.

A Summary of Observations

There is a large variance in the download-speeds in performing the same request multiple times. The variance is much higher when downloading from Cloudian than from AWS. We are not able to pinpoint the infrastructure component responsible for that.

We can expect a 30% reduction in the storage footprint of SRA objects to be achieved by upgrading the compression algorithm used by the current VDB library. 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.

The processing speed is correlated with the source being AWS or Cloudian. On average fastq-dump conversions from AWS are about 20% faster and have less variance. Avoiding partial downloads improves the overall processing speed, even though the tool is written in Python and does not use parallel processing.

When using VDB to access CSRA, client-side caching seems vital, especially for bigger archives, and does cost disk space. We were able to achieve the same download time as VDB (with caching), in our Python client using group-blob files (without any caching).

We did not observe significant differences in the speed of partial downloads from a single file compared to full-file downloads of the same data represented as a set of group-blob files.

The download speed is correlated with the source being AWS or Cloudian. On average downloads from AWS are about 2.5 times faster.

Partial downloads in sequential order (increasing file offset) do not differ in speed from downloading the same data-ranges in random order. Server-side caching is one possible explanation.

Parallelizing downloads of group-blob-files does not seem to improve the overall time significantly. This may be due to using multi-threading in Python, or a rate-limiter by IP-address in the infrastructure.