Final Report

CSCI 5900- Independent Study

Sagar Pathare

Date: April 26, 2022

Title: Using compression algorithms in GIGGLE to reduce disk space usage

1. Compression of the B+ tree leaves

Motivation

The large genomic data sets used by GIGGLE occupy a large amount of disk space. Could we use compression algorithms to reduce disk space utilization without significantly affecting the runtime?

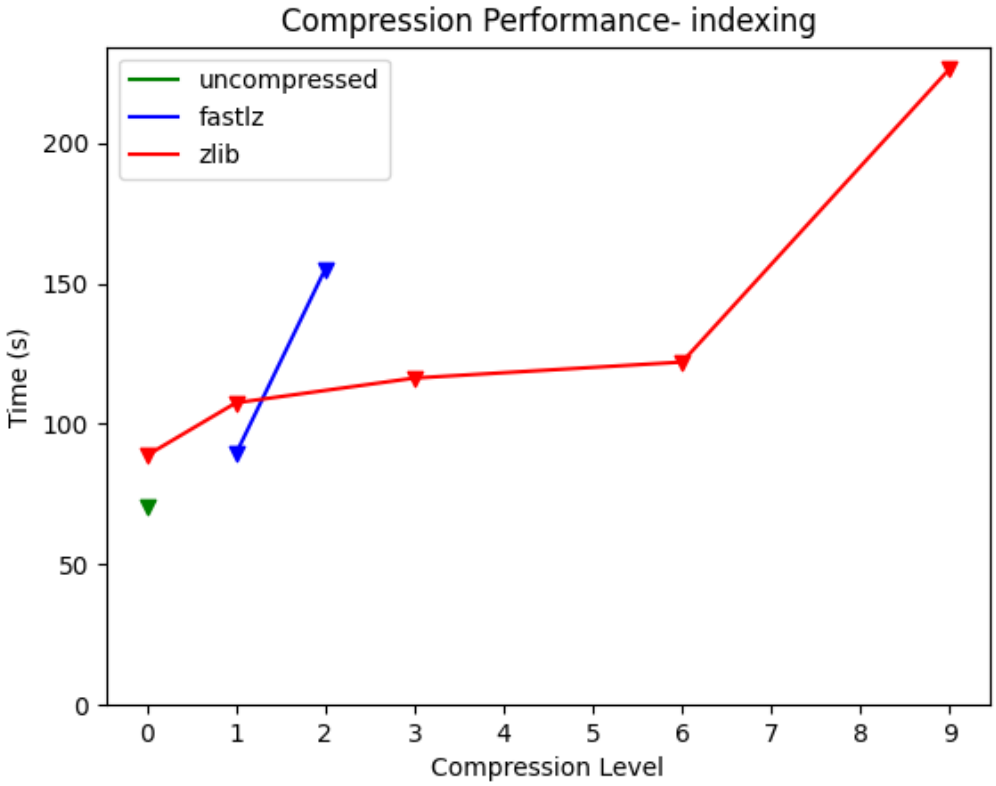
Method/implementation

We used the most common library for data compression in C- zlib. We created a wrapper/interface around the zlib functions for improved usability. We applied compression to the data stored in the data file. In addition to the existing data, we stored the uncompressed sizes in the index file. We added a file header/marker for both index and data files. The file header includes information about the compression method, compression level, and an extra flag reserved for future use. Files without headers from previous versions are assumed to be uncompressed and are read accordingly, thus enabling backward compatibility. We used function pointers to dynamically set the compress/uncompress functions, depending on the compression method mentioned in the file header.

Results

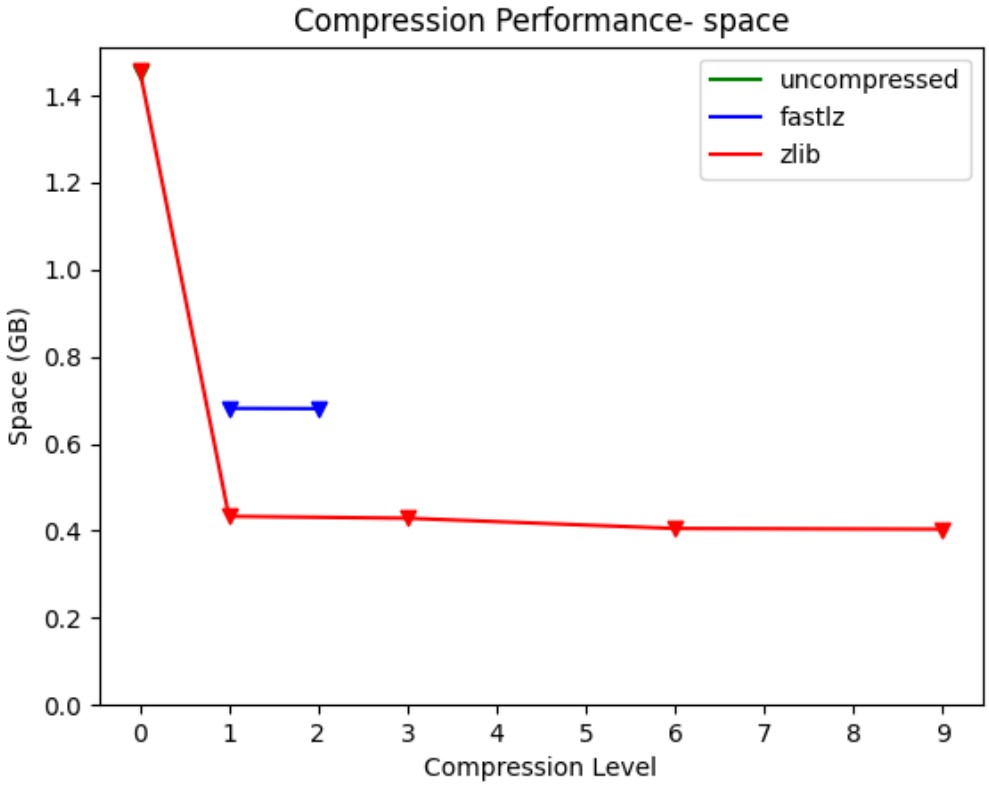
Time taken for Indexing (seconds)

type	time	user	system
uncompressed	70.49318158458	64.95069978	5.52916806
fastlz1	89.33131506806	82.75897032	6.56374108
fastlz2	155.0417450156	128.78304566	20.595867
zlib0	88.7408512163	78.20686414	10.524262
zlib1	107.49964638396	99.16664924	8.3172525
zlib3	116.30003864952	106.35132674	8.88081164
zlib6	122.00174653288	114.55789172	7.40855448
zlib9	226.4432716358	214.986536	11.37603618



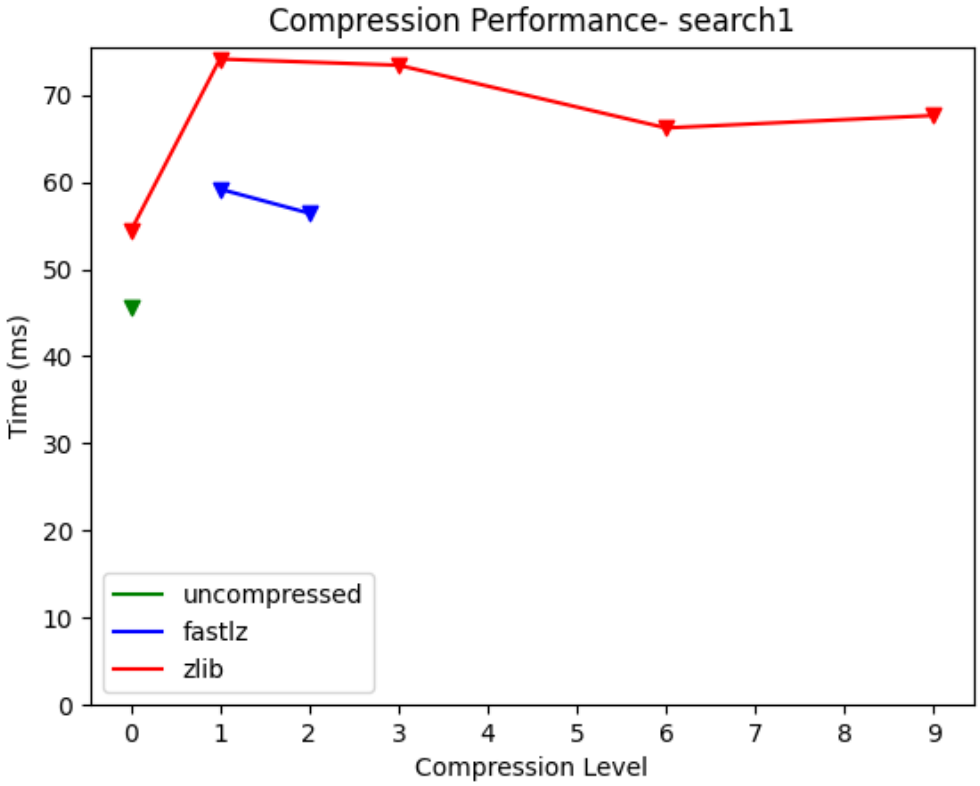
Disk Space Utilization (GB)

type	space
uncompressed	1.452596008
fastlz1	0.681222584
fastlz2	0.68060137
zlib0	1.459414745
zlib1	0.433347471
zlib3	0.428881416
zlib6	0.405332483
zlib9	0.40376754



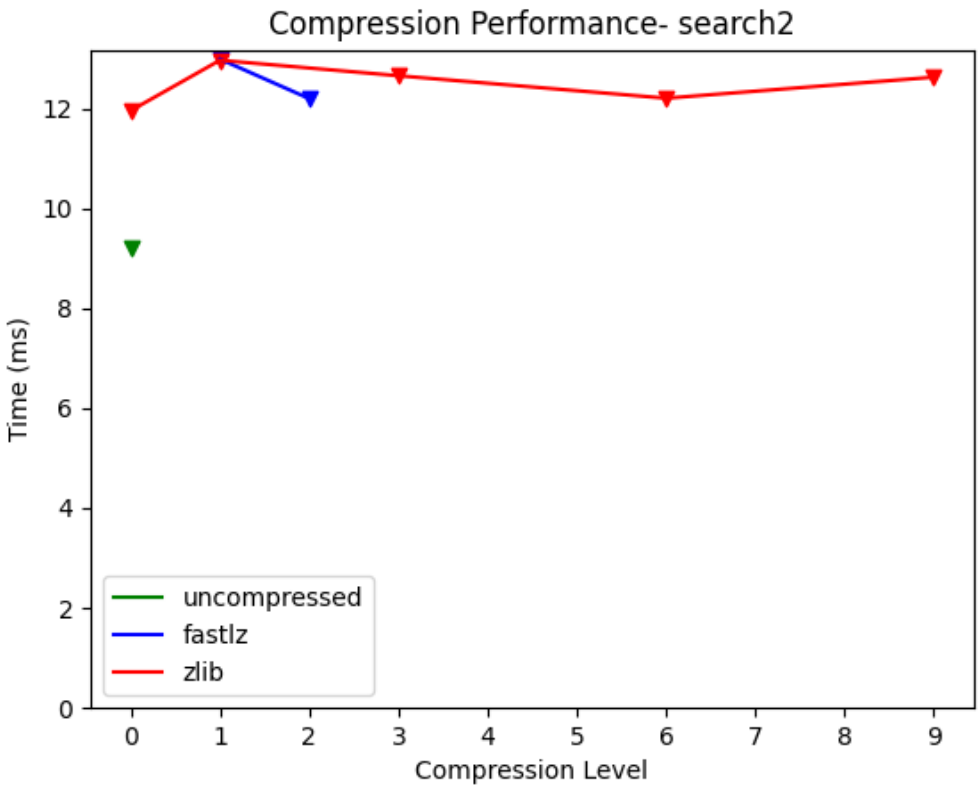
Time taken for Search 1 (milliseconds)

command	mean	stddev	median	user	system	min	max
uncompressed	45.5873693	3.959420647	44.7290814	31.80930286	13.75768714	40.9441094	62.0611574
fastlz1	59.18466978	5.09236337	59.78575998	37.83761843	21.31594118	51.78323798	74.08010998
fastlz2	56.41575026	3.620396294	55.79629656	37.70584	18.697718	52.23282456	77.13510456
zlib0	54.34283227	3.636371739	53.8785227	37.09720143	17.08244929	49.7197372	68.0634982
zlib1	74.10076984	6.043988274	71.65701646	55.84351846	18.24804923	67.48732846	92.28525846
zlib3	73.39287504	4.622238952	72.43769124	53.72295818	19.64546818	66.57508224	83.36368524
zlib6	66.19744482	2.242995965	65.84001782	49.64041217	16.5452913	63.34766232	74.41526232
zlib9	67.62356376	1.623330847	67.41949892	52.68799767	14.9192093	65.11356992	71.14124392



Time taken for Search 2 (milliseconds)

command	mean	stddev	median	user	system	min	max
uncompressed	9.189936953	0.5627173893	9.0503129	4.956227092	4.23508539	8.2692079	10.6013019
fastlz1	12.98639579	1.471285836	12.47858378	4.174342933	8.823754489	11.38921178	19.59861078
fastlz2	12.19613769	0.5875847746	12.03119652	3.905453208	8.274995	11.21100202	13.77797102
zlib0	11.95358581	0.7256513148	11.8664233	4.200792825	7.762784753	10.7087693	14.7913433
zlib1	12.96278894	0.9137541321	12.78595442	5.075546634	7.891110634	11.84051342	17.86654842
zlib3	12.6532173	0.64123711	12.4734273	5.266293897	7.37788507	11.6403703	15.2530783
zlib6	12.20321933	0.5730424908	12.02490832	5.261854609	6.953686435	11.33335482	14.26628082
zlib9	12.62452388	0.5751466442	12.48667264	5.127496946	7.492615961	11.60951164	14.42773764



Note: The time duration values are approximate as they are affected by other applications running in the background.

Discussion

The disk space usage was reduced by 70-72%, but the time taken for the search queries increased by 25-50%. We can conclude that fastlz2 is the best candidate. Compared to the uncompressed version, it reduced the space by around 53%, increased the search times by only 24% and 33% respectively. The indexing time was roughly 2.2 times, which is fine for a one-time task.

2. Compression of the offset index

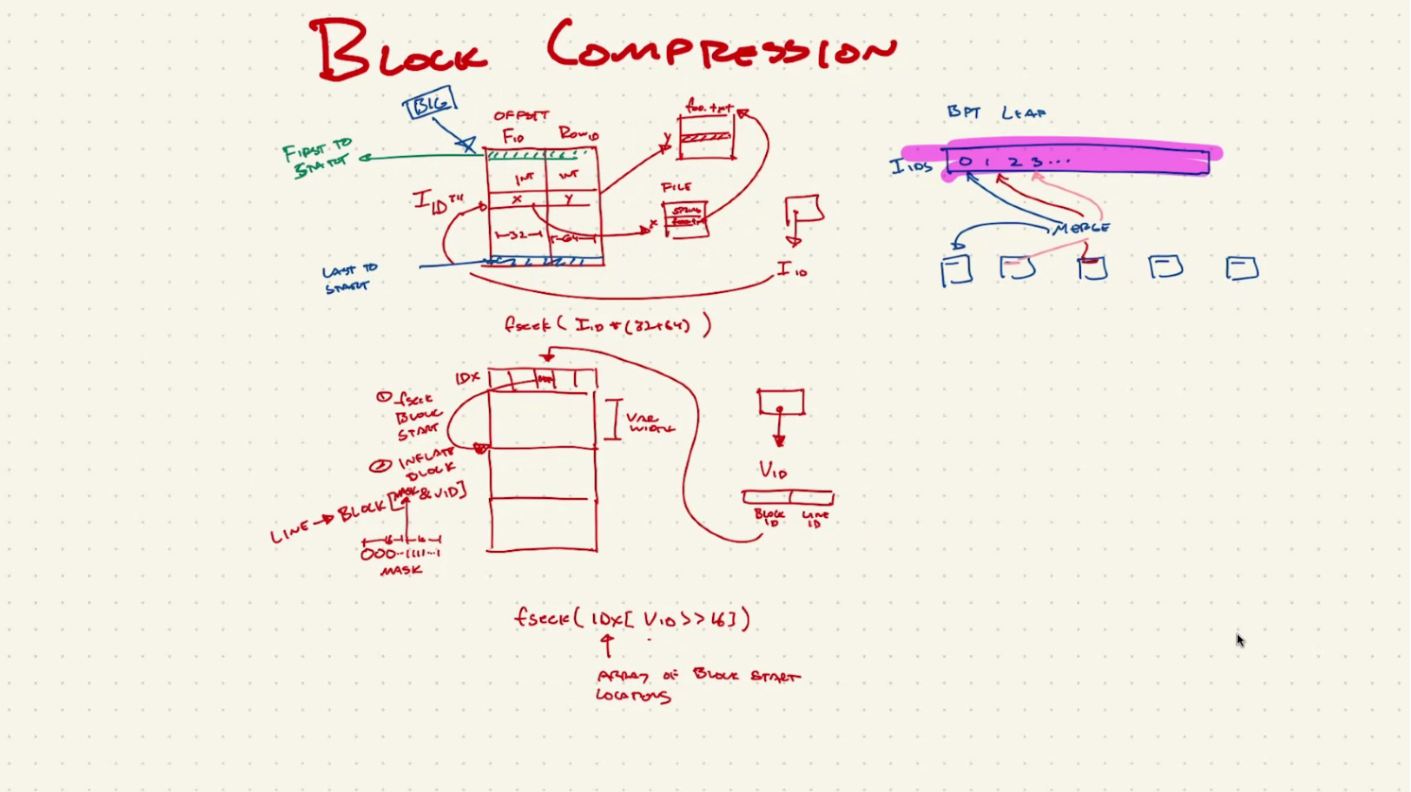
After compressing the files for leaves, currently, the file that takes up the most space is the offset index file- `offset_index.dat`. It stores a list of a pair of integers `x` and `y` where `x` is the file ID and `y` is the line ID in that file. We need the ability to read a particular position in the compressed file. After exploring various ways to implement compression, we concluded that we should implement block impression.

Structure of the `offset_index.dat` file

- 0-63 (8B) : `num`- the number of `file_id_offset_pairs` stored in the file
- 64-95 (4B) : `width`- the width of each `file_id_offset_pair`, currently 96 bits (12B)
    - Each `file_id_offset_pair` stores the two integers mentioned above
        - 32 bit `file_id` (`x`)
        - 64 bit `offset` (`y`)
- 96-... : `file_id_offset_pairs`

Block Compression Implementation

In the following diagram, Ryan has explained well how block compression could be implemented for `offset_index.dat`.



Block Compression- Ryan Layer

1. We will store a fixed number of pairs of integers as one block and then apply the compression on each block. What a good block size could be needs to be determined experimentally.
2. Currently, the leaf node stores `I_10`- which is the `offset_id` in the `offset_index.dat` file. Instead of storing `I_10`, we will store two values- the compressed block offset ID and the line offset ID within that block.
3. We will also store the compressed offsets for each compressed block.
4. To retrieve the original uncompressed `I_10`/`offset_id`, we will seek compressed file to the compressed block offset ID. We will uncompress the whole block and then seek to the line offset ID.

Proposed structure of the `offset_index_compressed.dat` file

- 0-63 (8B) : `num`- the number of `file_id_offset_pairs` stored in the file
- 64-95 (4B) : `width`- the width of each `file_id_offset_pair`, currently 96 bits (12B)
- 96-127 (4B) : `block_size`- the size of each uncompressed block- same for all blocks
- 128-... : `compressed_offsets`- the offset of each compressed block
- ...-... : `compressed_data`- the actual compressed blocks

Proposed Block Compression Implementation

We will create a standalone program called `block_compression.c` with the following functionality-

1. Compress `offset_index.dat` to `offset_index_compressed.dat`
2. Uncompress `offset_index_compressed.dat` to `offset_index.dat`
3. Read `offset_id` from `offset_index.dat`
4. Read `offset_id` from `offset_index_compressed.dat`

We need to write unit tests to make sure the output from 3 and 4 are the same.

To be explained later

- `mmap` usage
- `offset_data_append_data` function pointer usage instead of `fwrite`
- `OFFSET_INDEX_DATA` macro definition

Blocker (To do for Ryan)

- Figure out how to store two values in the leaf store

3. Other fixes made in the repository

- Added data files and executable files in the unit tests to gitignore.
- Fixed the unit tests Makefile issue that prevented two consecutive builds without running make clean- the _Runner.c files were also being considered for tests. Added wildcard filter to ignore them.
- Fixed minor bugs and formatting.