FFIndex: Compressed Indexing for Parallel FAST-Q Parsing

Team Members:
Siddhant Bharti
Prajwal Singhania
Rakrish Dhakal

Project Description

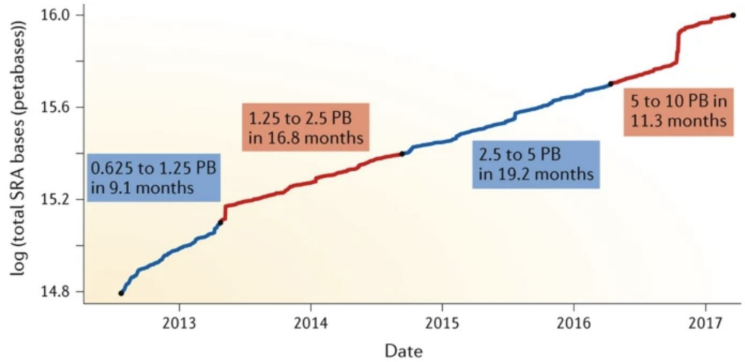
- Our project's aim is to address the challenge of efficiently parsing compressed FASTQ files. This is critical for genomic data processing pipelines but is hindered by the sequential nature of current decompression and parsing methods.

- The goal is to develop a tool and accompanying library that facilitates parallel processing by creating and utilizing an index over compressed FASTQ files.

Motivation

- Sequence Read Archive (SRA) is predicted to double every 12-18 months. A solution that allows for parallel reading of compressed FASTQ files would facilitating research that relies on large-scale genomic datasets[1].
- Most files hosted on SRA are not compressed in blocks[2].
- If multiple files are to be read in an an application, a file-parallel solution can work.
- What about a single file though?

Figure 1: Increase in storage of next-generation sequencing data.



Motivation

- Single FASTQ files used in large sequencing data experiments can be extremely large. For example, the size of the BGISEQ (MGISEQ-2000RS)[2] FASTQ file is around 37GBs.

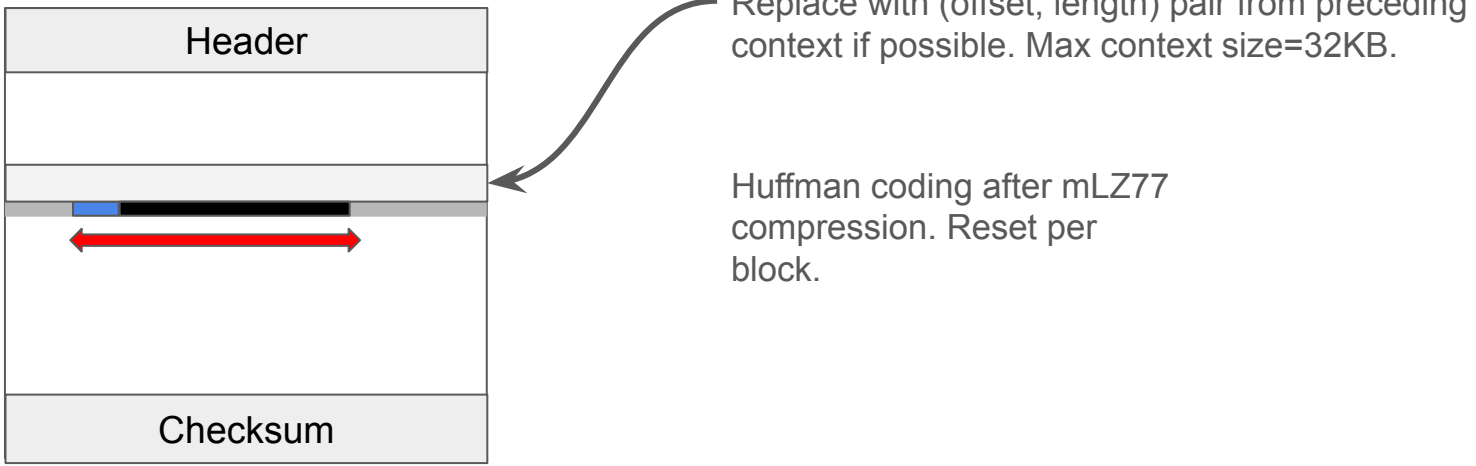
- gunzip can read compressed data at around 30-50 MB/sec which is order of magnitude less than read throughput of current SATA/NVMe solid-state drives[2].

```
[sbharti@sfeiziwks01 CMSC701-Project]$ ./countbases.out data/nematode.fq.gzip | grep -E "Time"
Time taken (to read): 16106 milliseconds
Time taken (total): 25594 milliseconds
[sbharti@sfeiziwks01 CMSC701-Project]$
```

- At this rate, doing analysis for large gzip files can be time and cost-prohibitive.
- **A clever solution to decompress gzip files in parallel is needed.**

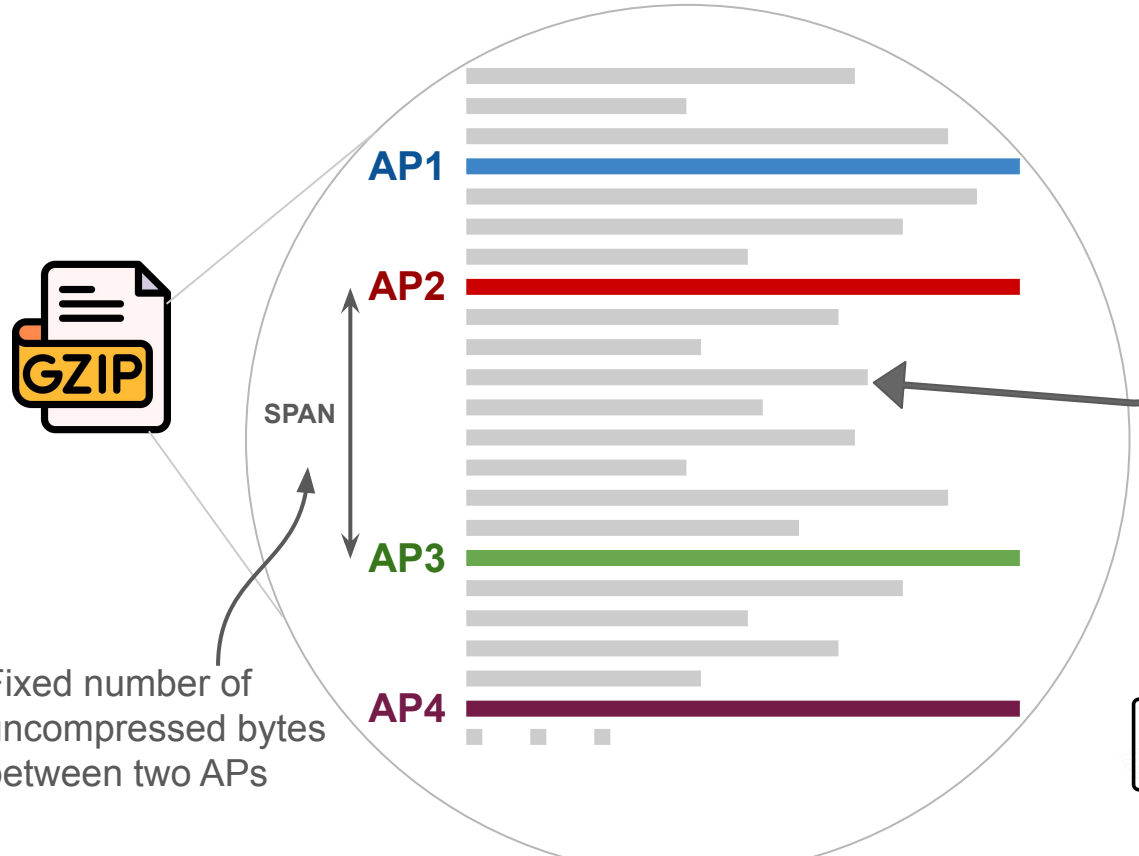
Random Access to GZIP files?

- Gzip achieves best size reduction for FASTQ files. However, gzip's DEFLATE (mLZ77 & huffman coding) algorithm is **sequential**. This makes **random access** in compressed FASTQ files[1] hard.



- But is it impossible? NO!

zran.c:[3] Example in zlib that creates “access points” for fast random access to gzip



If we want to read from here, we don’t have to start from the beginning! (can start from **AP2**)

But zran just creates the index, uses it for random access **one time**…

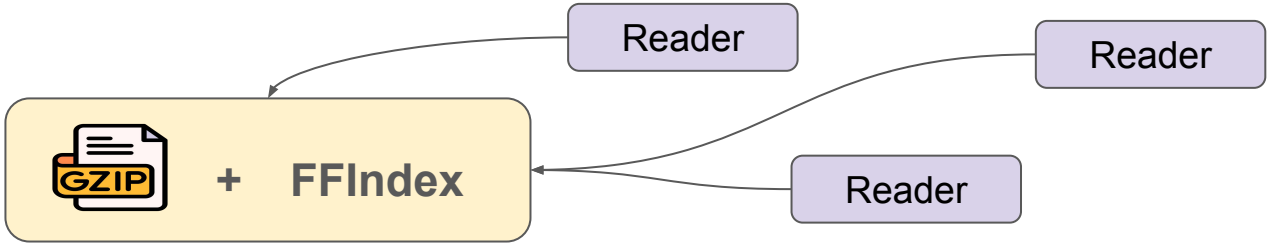
We need to **save it to disk** for reuse in the future!

Fixed number of uncompressed bytes between two APs

AP1 **AP2** **AP3** **AP4** …

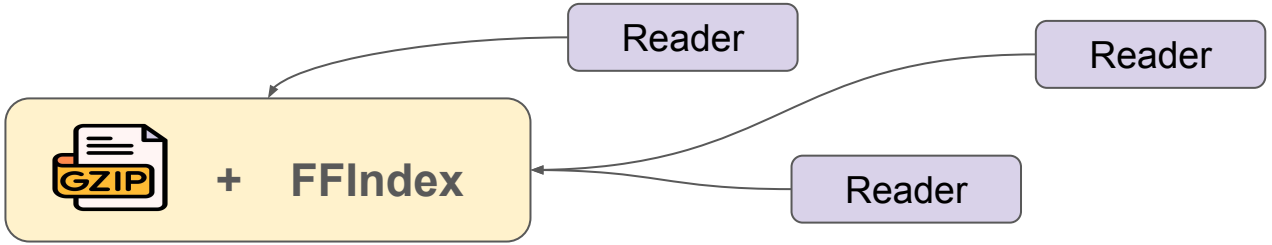
Our approach

- **Index Building**:
  - Modify zran's index specifically for FASTQ files.
  - Save these access points and record boundaries to disk to supplement the original GZIP.
- **Parallel Parser Implementation**:
  - Implement producer-consumer model for parsing single FAST-Q files in parallel
  - Use saved index for parallel reading, using zran's random access algorithm and information on record boundaries.
  - Benchmark speedup and memory tradeoffs for various SPANs and other parameters.

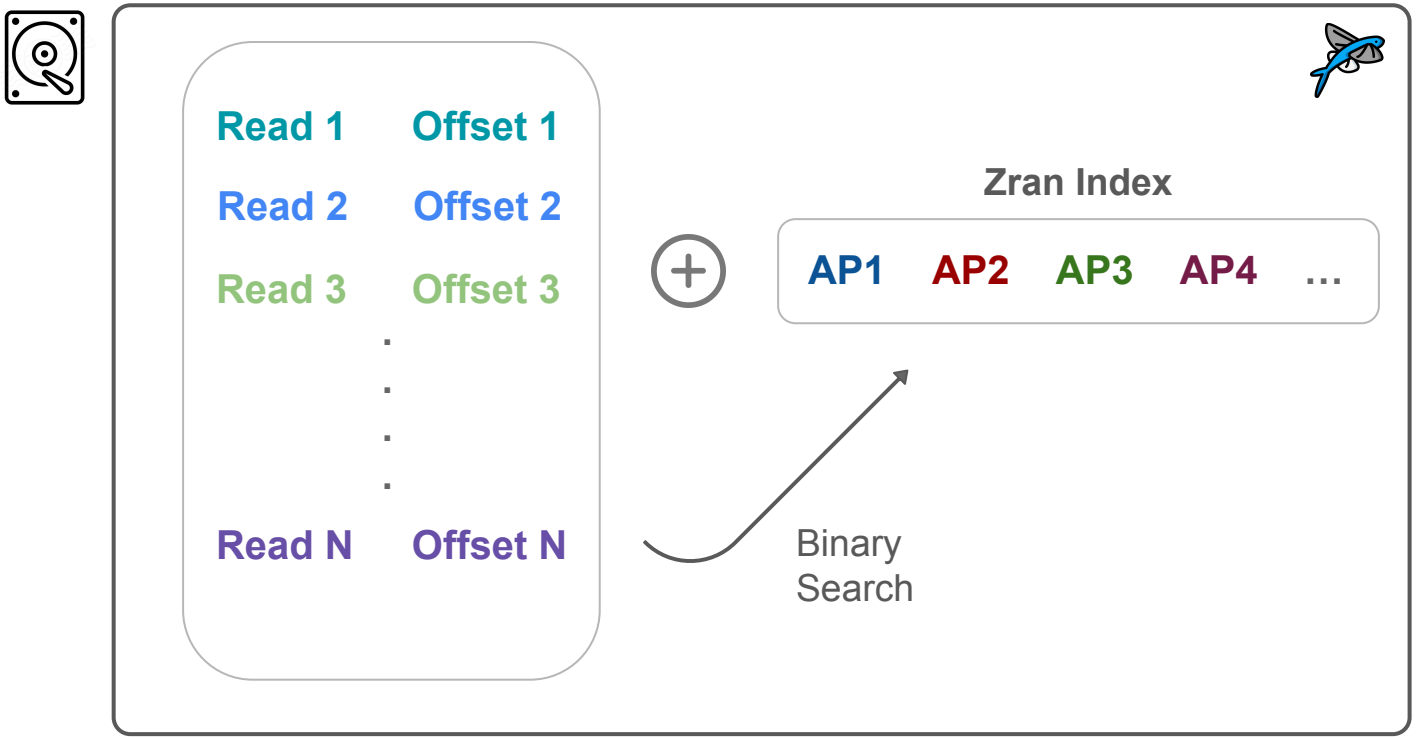


Our approach

- **Index Building**:
  - Modify zran's index specifically for FASTQ files.
  - Save these access points and record boundaries to disk to supplement the original GZIP.
- **Parallel Parser Implementation**:
  - Implement producer-consumer model for parsing single FAST-Q files in parallel
  - Use saved index for parallel reading, using zran's random access algorithm and information on record boundaries.
  - Benchmark speedup and memory tradeoffs for various SPANs and other parameters.

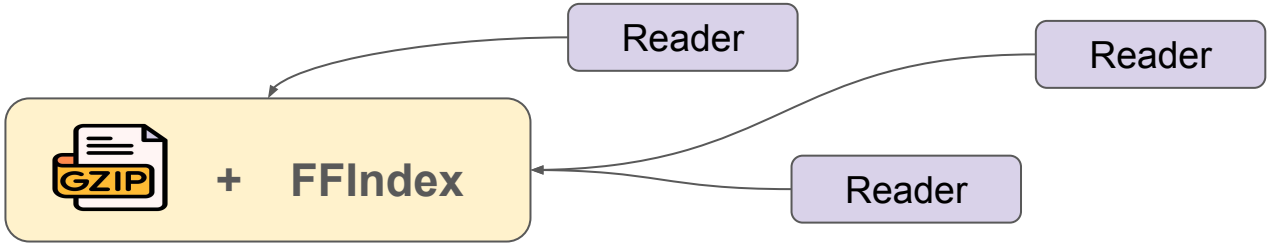


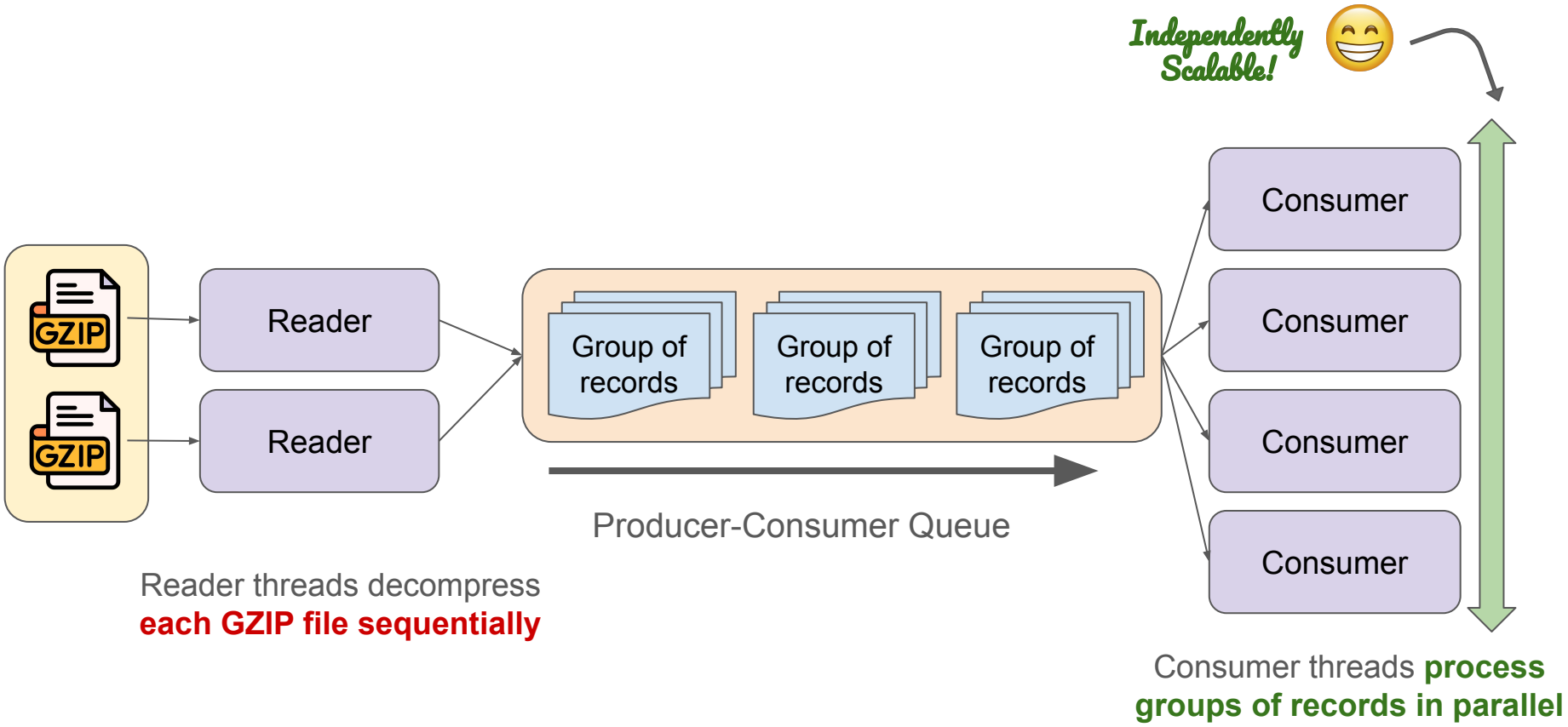
Our approach: FFIndex

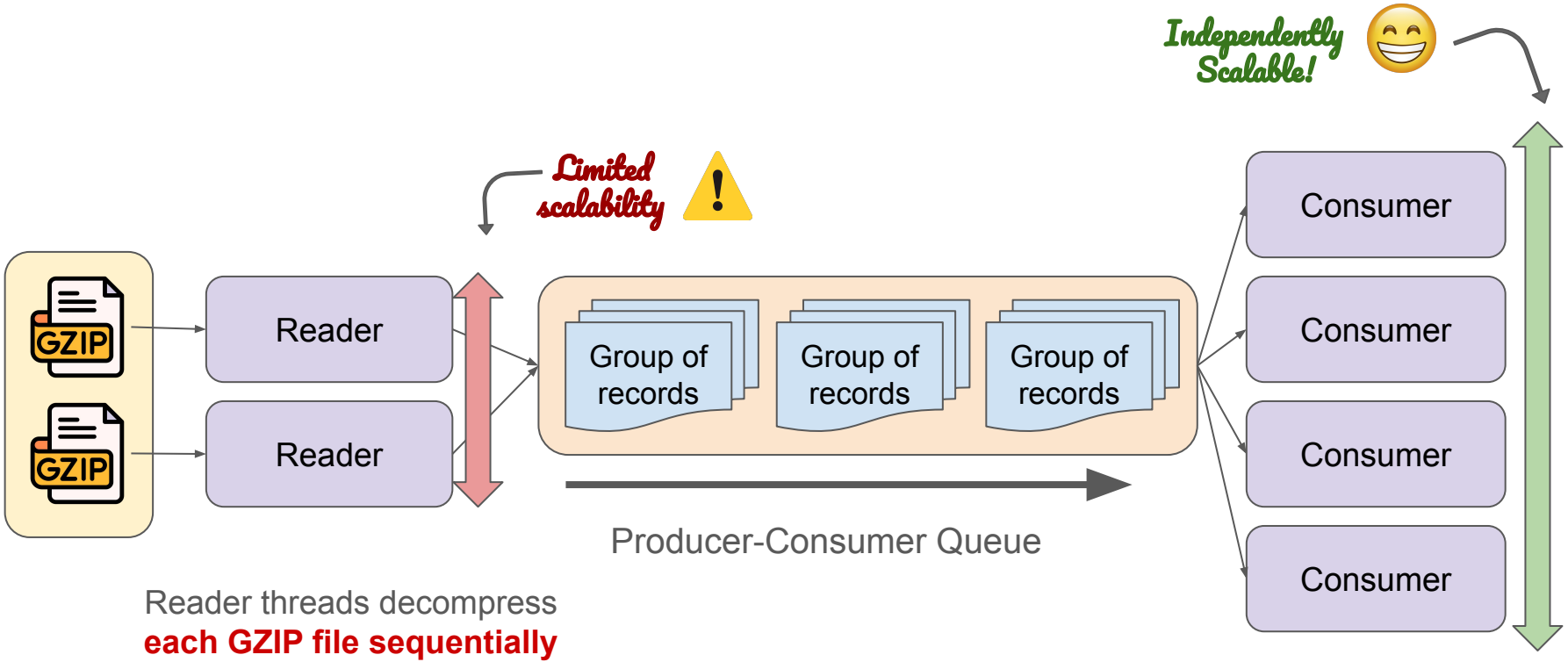


Our approach

- **Index Building**:
  - Modify zran's index specifically for FASTQ files.
  - Save these access points and record boundaries to disk to supplement the original GZIP.
- **Parallel Parser Implementation**:
  - Implement producer-consumer model for parsing single FAST-Q files in parallel
  - Use saved index for parallel reading, using zran's random access algorithm and information on record boundaries.
  - Benchmark speedup and memory tradeoffs for various SPANs and other parameters.




Independently Scalable!
GZIP
GZIP
Reader
Reader
Group of records
Group of records
Group of records
Consumer
Consumer
Consumer
Consumer
Producer-Consumer Queue
Reader threads decompress
each GZIP file sequentially
Consumer threads process
groups of records in parallel

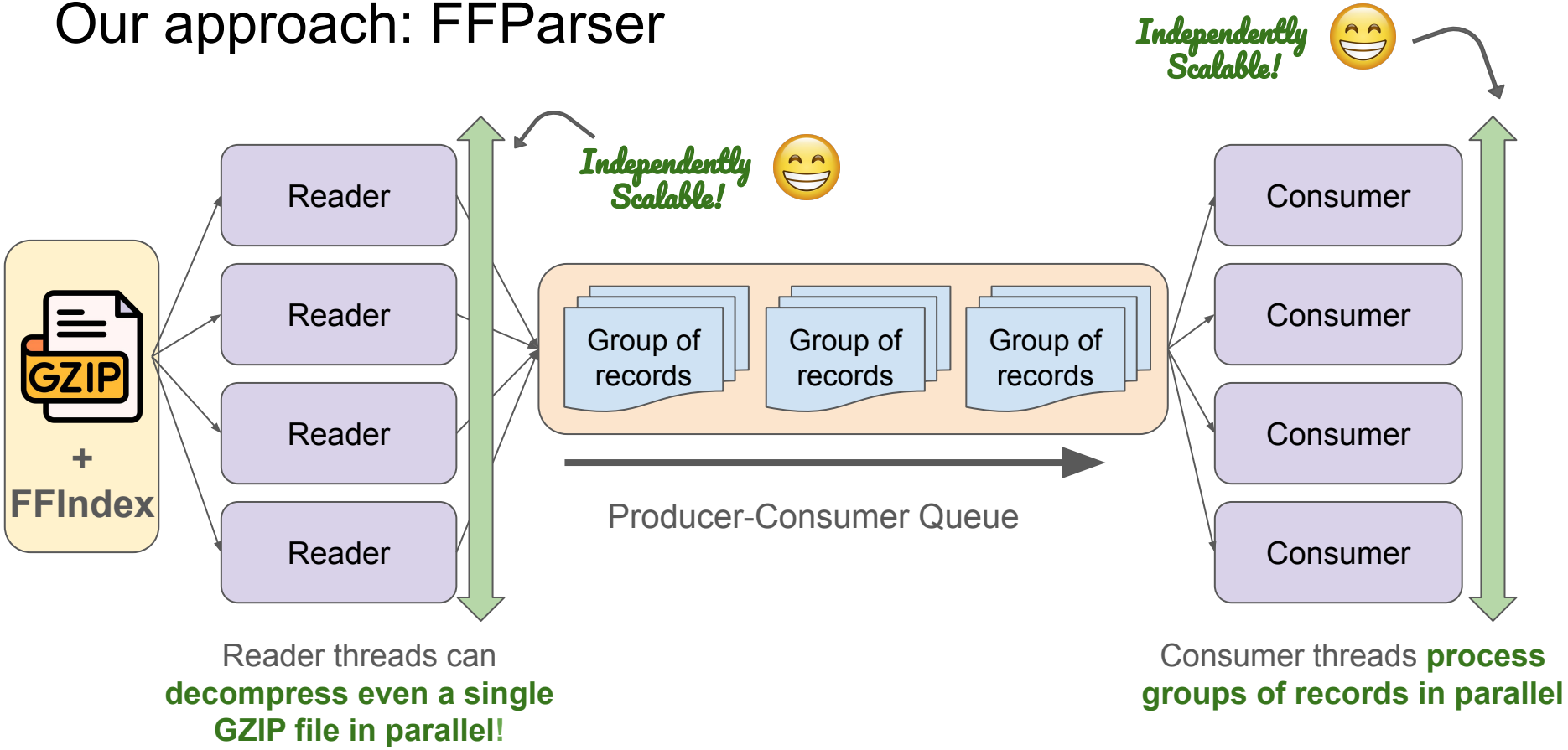


Reader threads decompress
each GZIP file sequentially

…can we do better?

Consumer threads **process groups of records in parallel**

Our approach: FFParser



Evaluation Plan

- **How lightweight is the index?**
  - Assess memory usage, storage needs for the index compared to the original files

- **Parallel Performance**
  - Evaluate strong scaling performance to see how much parallelism helps in reducing the time for parsing

Preliminary Statistics on Index*

SPAN = 1048576

	Compressed FASTQ file size	Compressed Index file size	Time to create the Index	Time to read the Index
Salmonella	1.4 MB	39 KB	56 ms	< 1ms
Ecoli	39 MB	5 MB	2819 ms	58 ms
Dataset C - Ilumina (seqkit benchmark dataset)	520 MB	38 MB	18846 ms	315 ms
Nematode	655 MB	48 MB	25816 ms	400 ms

* run on M2 Pro Macbook

Preliminary Statistics on Index*

SPAN = 1048576

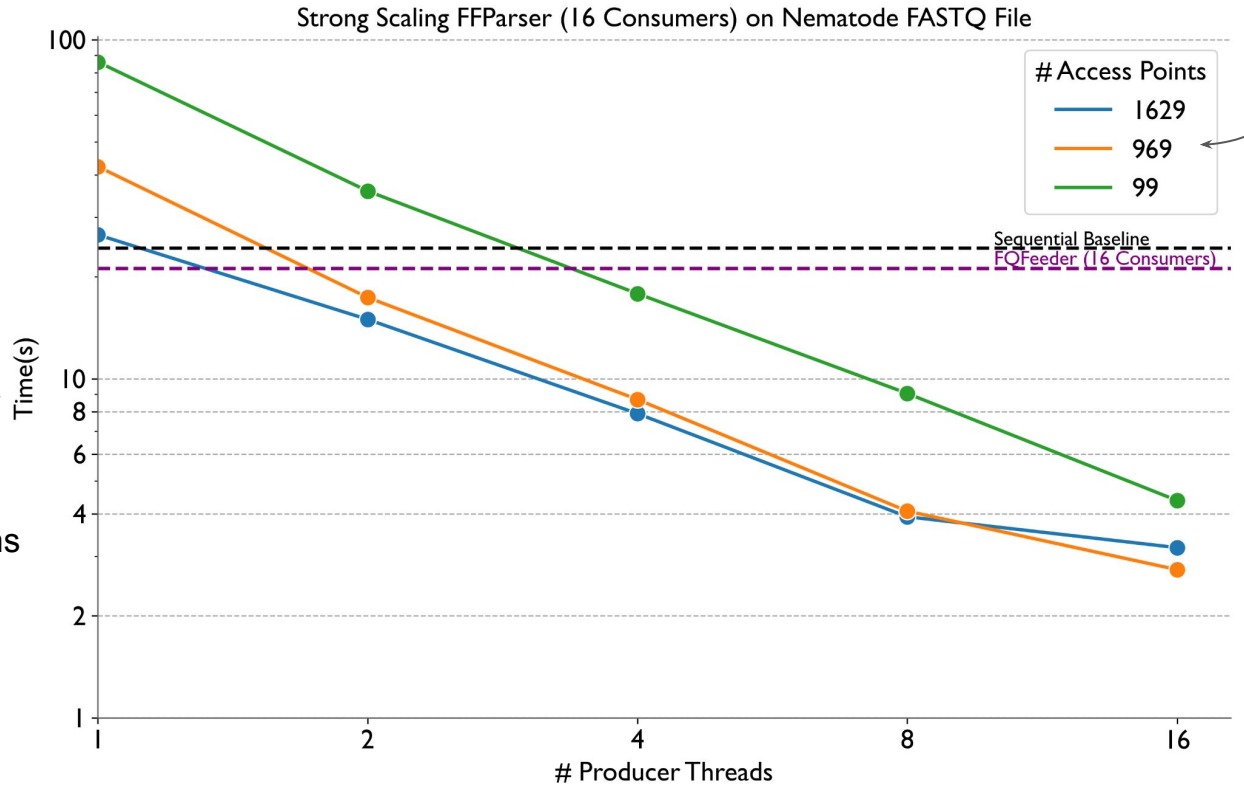
	Compressed FASTQ file size	Compressed Index file size	Time to create the Index	Time to read the Index
Salmonella	1.4 MB	39 KB	56 ms	< 1ms
Ecoli	39 MB	5 MB	2819 ms	58 ms
Dataset C - Ilumina (seqkit benchmark dataset)	520 MB	38 MB	18846 ms	315 ms
Nematode	655 MB	48 MB	25816 ms	400 ms

8.8%!

* run on M2 Pro Macbook

Preliminary Parallel Performance Results*

Access Points ∝ 1/SPAN



Average of 5 Runs

* run on Zaratan's Compute Node

Further Evaluation and Optimizations

- **Datasets:**
  - Repeat experiments for bigger FAST-Q files

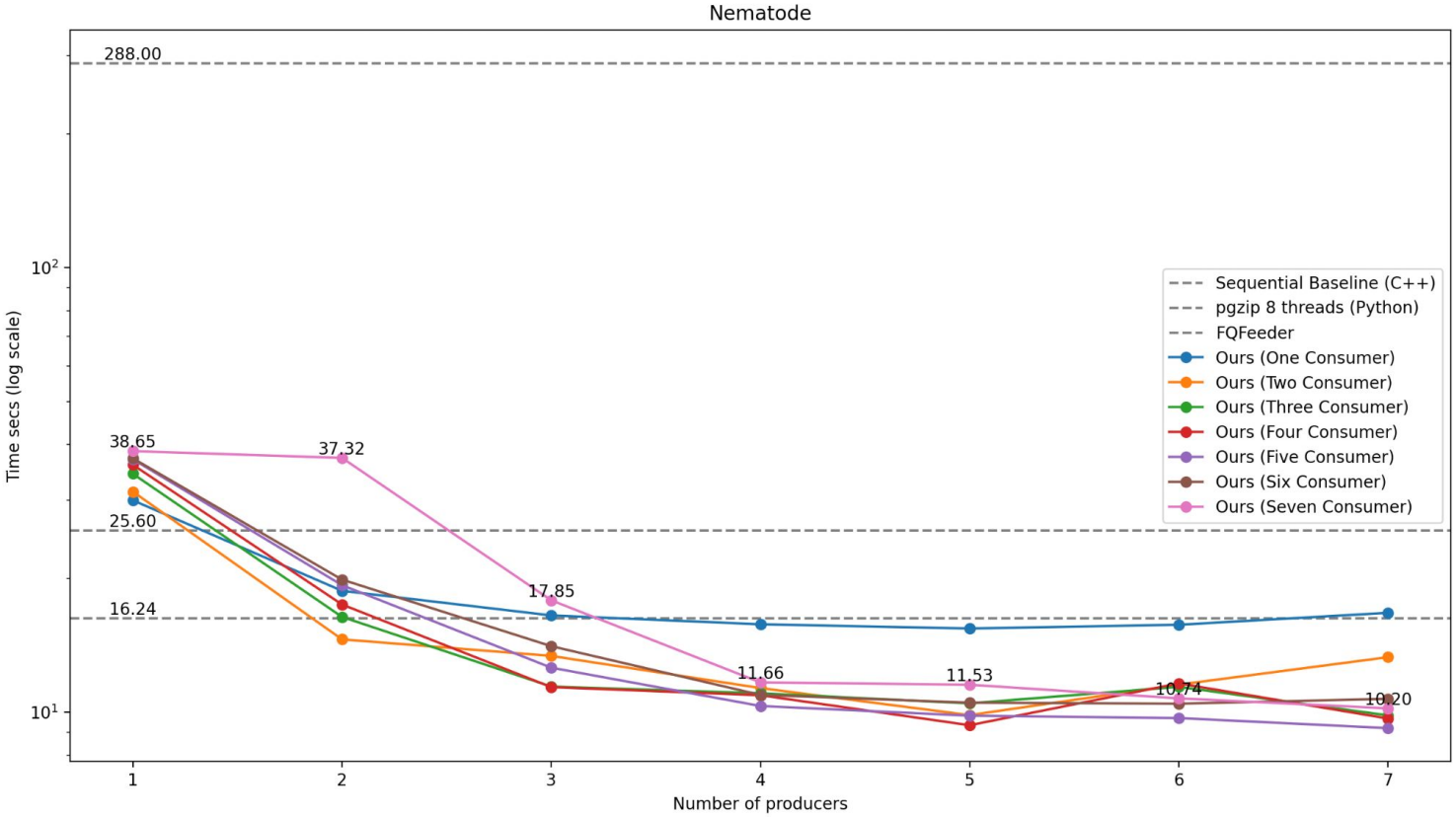
- **Component-wise Breakdown:**
  - Analyze which parts of the index are most space-consuming and target optimizations
  - Analyse what part of the codes can be optimised via Trace Analysis to identify bottlenecks and improving scaling efficiency

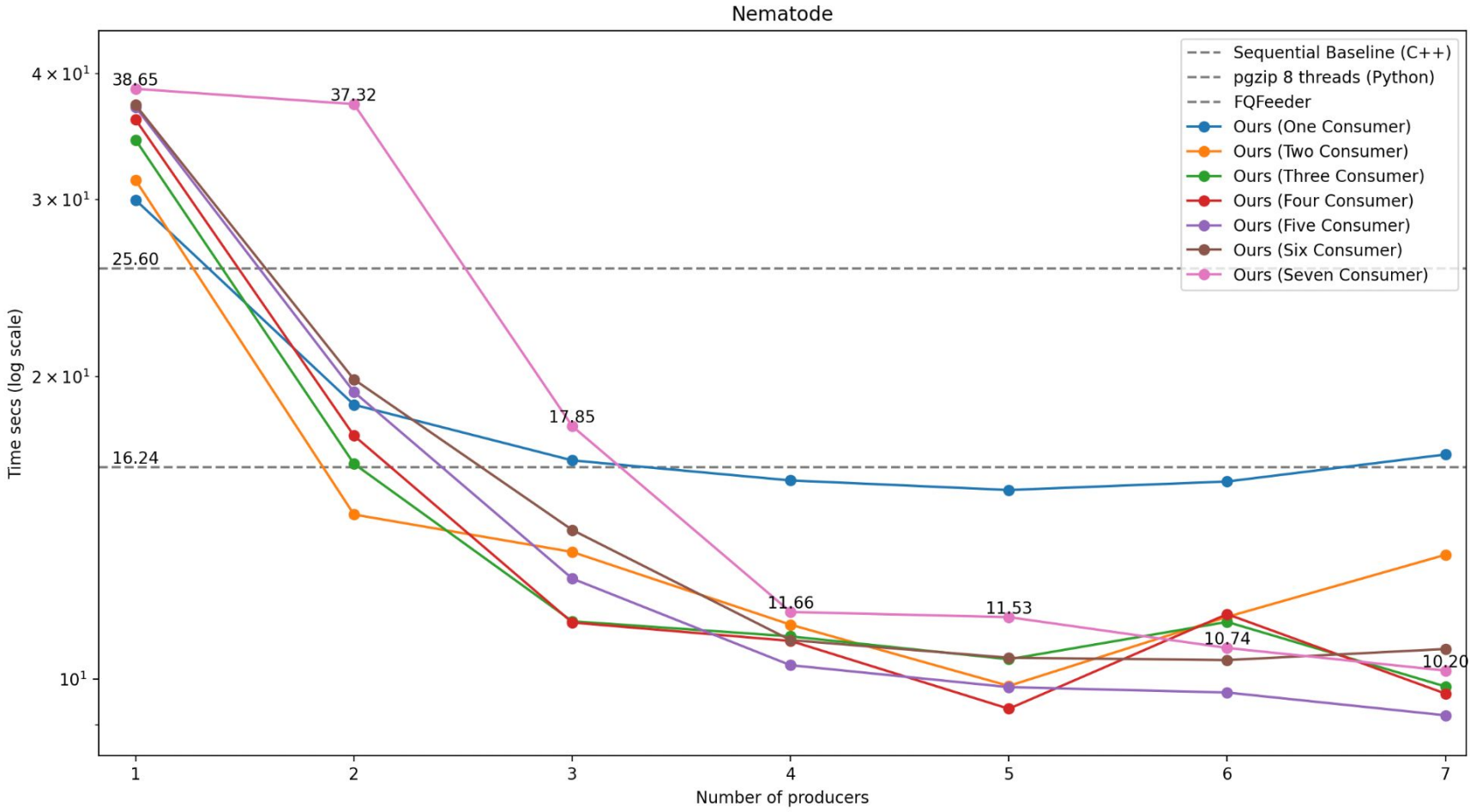
- **Optimizations Possible:**
  - Currently we store uncompressed byte offset for all FASTQ records. Probably only need to store for blocks of N records

Questions?

Evaluation Plan

- Lightweight Index:
  - Assess memory usage, storage needs for the index.
- Parallel Performance:
  - Evaluate concurrency, throughput, and resource efficiency.
- Datasets:
  - Try with bigger dataset.
- Component Breakdown:
  - Analyze which parts of the index are most space-consuming and target optimizations.
- Trace Analysis:
  - Identify bottlenecks to enhance parsing speed and efficiency.


Nematode
288.00
25.60
16.24
10^2
10^1
Time secs (log scale)
38.65
37.32
17.85
11.66
11.53
10.74
10.20
1
2
3
4
5
6
7
Number of producers
Sequential Baseline (C++)
pgzip 8 threads (Python)
FQFeeder
Ours (One Consumer)
Ours (Two Consumer)
Ours (Three Consumer)
Ours (Four Consumer)
Ours (Five Consumer)
Ours (Six Consumer)
Ours (Seven Consumer)


Nematode
Sequential Baseline (C++)
pgzip 8 threads (Python)
FQFeeder
Ours (One Consumer)
Ours (Two Consumer)
Ours (Three Consumer)
Ours (Four Consumer)
Ours (Five Consumer)
Ours (Six Consumer)
Ours (Seven Consumer)
38.65
37.32
25.60
17.85
16.24
11.66
11.53
10.74
10.20
Time secs (log scale)
Number of producers