Project Proposal

CERN-HSF, Compiler Research

Enhance and Develop GeneROOT Infrastructure

Mentors

- Vassil Vassilev (vvasilev@cern.ch)
- Martin Vassilev (mvassilev@uni-plovdiv.bg)

Personal Details

Jeffrey Zhang

Bachelor of Science, Physics,
Nagoya University, Japan

Phone: +81 080 6385 3134
Github: https://github.com/RedBlueBird
Email: jiefu.zhang1226@gmail.com
School Email: zhang.jeffrey.v4@s.mail.nagoya-u.ac.jp

Contents

PROJECT DETAILS

Introduction

Large-scale biological data, such as a fully sequenced human genome, typically occupies ~500 GB. Analyzing such datasets for research involves data volumes that exceed petabytes. Handling data at this scale requires a highly robust underlying software infrastructure. To meet this challenge, the GeneROOT project draws on CERN's extensive expertise in managing massive physics datasets through its columnar-based ROOT software framework. The GeneROOT project aims to adapt this framework specifically for processing biological data.

During the 2025 GeneROOT GSoC project, Aditya Pandey established the RNTuple data model for genome sequences. It currently supports region queries, conversion from SAM to RNTuple, and a benchmark comparison against the industry-standard CRAM format on a single test sample `HG00154` from the 1000 Genomes Project.

However, the results reveal several limitations. The benchmark suite relies on a single low-coverage sample with hard-coded file paths, which is insufficient for a credible comparison with tools such as SAMtools and CRAM. In terms of performance, RNTuple's index lookup itself performs a linear scan that does not scale to production-sized datasets. In terms of functionality, RAMtools cannot currently export records back to SAM, has no merge operation to complement the chromosome splitter, no sort, and no statistics tools. These gaps leave RAMtools as a proof of concept rather than a usable pipeline component.

This proposal aims to build on the foundation established in GSoC 2025 by expanding the benchmark suite, optimizing core performance, and introducing new functionalities.

Deliverables

Specifically, I plan to complete the following deliverables

1. Benchmark on heavy bioinformatics datasets
   (a) Refactor the existing benchmark suite
   (b) Identify quality bioinformatic datasets such as `HG001 – HG007`
   (c) Additional benchmark metrics such as memory usage
2. Benchmark against various file formats
   (a) Compare `SAM`, `RAM`, and `CRAM` through `system()` calls in all `benchmark/` scripts.
3. Improve data compression algorithms adapted for genomic sequences
   (a) Crumble, QVZ, CALQ, P-block, etc.
4. Optimize indexing and search capabilities for large genomic datasets
   (a) Fix the linear scan in `GetRowsInRange()`
   (b) Eliminate redundant `fIndexMap`
   (c) Configurable index granularity
   (d) Implement no-index query fallback
5. Add support for Stat, View, Split, Merge, and Sort
   (a) Functionalities mirror SAMtools

IMPLEMENTATION PLAN

Task 1: Benchmark on heavy bioinformatics datasets

Establishing an extensive and reliable benchmark suite is my primary priority. First, testing on larger datasets ensures statistical confidence; second, it can more accurately quantify performance gains from subsequent optimizations.

I have divided this task into three key phases:

- **Refactor the existing benchmark suite**

  Excluding `sam_to_ram_benchmark.cxx` for legacy TTree data model and `generate_sam_benchmark.cxx` for data generation, there are three scripts for benchmarking so far. One script `region_query_benchmark.cxx` references `HG00154` data sample through hardcoded file paths, as seen in line `21` below.

```
class RegionQueryFixture : public benchmark::Fixture {
public:
   void SetUp(const benchmark::State &state) override
   {
      region_idx_ = static_cast<int>(state.range(0));

      sam_file_ = "/media/aditya/213e0e46-6f86-4288-8b79-74851c34314f/
         output_big.sam";
      ttree_root_file_ = "/media/aditya/213e0e46-6f86-4288-8b79-74851
         c34314f/output_big_lzma.root";
      rntuple_root_file_ = "/media/aditya/213e0e46-6f86-4288-8b79-74851
         c34314f/output_root.root";
   }
}
```

  The other two scripts `conversion_time_benchmark.cxx` and `chromosome_split_benchmark.cxx` do not use actual data samples, but instead rely on pseudo-generated data from `GenerateSAMFile()` for performance testing.

  In addition, the current benchmark suite uses Google Benchmark, which supports JSON output, but achieving this requires running a separate command for each benchmark script, as running `cmake --build build --target benchmark` does not support this feature.

  To address those issues, first, I will create `benchmark/benchmark_config.h` to introduce command-line argument parsing for dataset file paths (fallback to `GenerateSAMFile()` if none listed), and incorporate this into all the benchmark scripts. Second, I will modify `benchmark/CMakeLists.txt` to allow all benchmark scripts to be run with custom outputs, such as JSON/CSV, in a single command using custom CLI flags.

- **Identify quality bioinformatic datasets**

  Two of the most popular bioinformatic datasets on humans are ”Genome in a Bottle (GIAB)” and ”1000 Genome Project (1KGP)”. Samples that have higher sequencing coverage, particularly from whole-genome sequencing platforms, are generally more widely used and trusted for benchmarking. Samples `HG001 - HG007` are among the most commonly cited, making them excellent starting points for this benchmark.

Although most bioinformatic benchmarks use mostly only human genome samples, non-human samples could also be worth potential consideration for demonstrating generalizability. The NCBI Datasets contain a wide variety of genome samples across the tree of life.

Since there is no overhead cost in switching between samples, I will maintain flexibility in selecting specific genome samples and finalize them in consultation with my mentors.

- **Additional Benchmark notes**

  Google Benchmark is a powerful tool, but it is currently underutilized inside `benchmark/`. Since GeneROOT aims to operate in high-performance computing environments in the future, memory usage/profiling would be a good metric to include: incorporate Google Benchmark's `MemoryManager`.

  Lastly, I will include documentation with `.ipynb, .md` files in a new folder `doc/` (which will be used for all subsequent documentation in the following tasks) on `benchmark/` to allow others to easily reproduce the results.

Task 2: Benchmark against various file formats

Once the RNTuple benchmark suite is established, I will incorporate other data formats as a baseline for performance comparisons.

There are two potential methods to do cross-format comparisons. The first one is to keep the existing method used in the benchmark directory, `chromosome_split_benchmark.cxx` calls `samtools` through `system()` to execute SAMtools functions (one example at line `60` shown below).

```
for (auto _ : state) {
  std::string bam_file = "bench_st_tmp.bam";
  std::string sorted_bam = "bench_st_sorted.bam";

  system(("samtools view -bS " + sam_file + " -o " + bam_file + " 2>/dev/
     null").c_str());

  system(("samtools sort " + bam_file + " -o " + sorted_bam + " 2>/dev/
     null").c_str());

  system(("samtools index " + sorted_bam + " 2>/dev/null").c_str());
```

The second method is inspired by a repository called `DuckDB`. In order to show their performance gains against other popular database systems like `Spark`, they created a separate repository to manage the scripts for different systems.

Given the limited time, extending the current `system()` call approach is the most practical solution, allowing for a future migration to an independent benchmarking repository if necessary.

Currently, only `chromosome_split_benchmark.cxx` incorporates `samtools` using `system()` commands. I will expand on this by modifying all the existing benchmark scripts to call `samtools` through `system()`, and check against the file formats `SAM, RAM, CRAM` one at a time, similar to how `chromosome_split_benchmark.cxx` currently operates.

Task 3: Improve data compression algorithms adapted for genomic sequences

Currently, when converting from SAM to RNTuple, data samples undergo two layers of compression, as described by the following table.

SAM Field	Layer 1 (RAMtools)	Layer 2 (ROOT, one global choice)
`QNAME`	none	ZSTD / LZ4 / LZMA
`FLAG`	none	ZSTD / LZ4 / LZMA
`REFID`	none	ZSTD / LZ4 / LZMA
`POS`	none	ZSTD / LZ4 / LZMA
`MAPQ`	none	ZSTD / LZ4 / LZMA
`CIGAR`	BAM `uint32` encoding (lossless)	ZSTD / LZ4 / LZMA
`REFNEXT`	none	ZSTD / LZ4 / LZMA
`PNEXT`	none	ZSTD / LZ4 / LZMA
`TLEN`	none	ZSTD / LZ4 / LZMA
`SEQ`	2-bit packing (lossless)	ZSTD / LZ4 / LZMA
`QUAL`	Phred33 / Illumina 8-bin / Drop	ZSTD / LZ4 / LZMA
`TAGS`	none	ZSTD / LZ4 / LZMA

RNTuple currently offers three compression modes: lossless Phred+33, Illumina 8-bin lossy binning, and full drop. While Illumina 8-binning provides a baseline for lossy compression, it relies on a simple rounding scheme. Evaluating modern genomic compression literature can yield more sophisticated algorithms.

I will evaluate the following compression algorithms for further study: Crumble (Bonfield 2019), QVZ (Malysa et al. 2015), CALQ (Voges et al. 2018), and P-block (Canovas et al. 2014).

Once I have evaluated and selected the optimal compression algorithms, I will introduce them by first extending the `EQualCompressionBits` enum in `RAMNTupleRecord.h` and the corresponding `EncodeQuality()`/`DecodeQuality()` functions in `RAMNTupleRecord.cxx` with new quality policies:

```
enum EQualCompressionBits {
   kPhred33         = 1 << 14,
   kIlluminaBinning = 1 << 15,
   kDrop            = 1 << 16,
   newAlgorithmHere = 1 << 17,
   ...
};
```

I will measure quantitative improvements through benchmarking by introducing a new bench-

mark, `qual_compression_benchmark.cxx`. For each new compression algorithm applied to `QUAL`, I will run it on the same datasets and compare file sizes against CRAM's lossy/lossless modes.

Finally, I will document the summarized genomic compression techniques, experimental results, and a recommended default policy for typical data samples.

Another area worth exploring is per-column compression. All the SAM field data are being converted to ROOT through one of three existing uniform general compression algorithms: ZSTD, LZ4, and LZMA. However, all the SAM fields have different entropies: for example, `REFID` values are mostly either `0` or `1`, but `SEQ` is mostly a string with less predictable patterns. Because different SAM fields exhibit distinct entropy profiles, applying a per-column compression tailored to each field would yield better compression ratios. However, the major problem behind this optimization gain is that ROOT, to my knowledge, does not support per-column-based compression. This could be something worth discussing in the future with ROOT.

Task 4: Optimize indexing and search capabilities for large genomic datasets

I plan to implement the following optimizations:

- **Fix the linear scan in `GetRowsInRange()`**

  The current implementation has a critical performance issue. `GetRowsInRange()` in `RAMNTupleRecord.cxx` performs a linear scan over (time complexity $\mathcal{O}(N)$) the entire index vector for every query.

  However, an inspection of the definition of `GetRow()` in the same file shows that they are similar, but the latter has a faster time complexity $\mathcal{O}(\log(N))$. A closer comparison reveals that `GetRowsInRange()` is fundamentally a generalization of `GetRow()`.

  I will redefine `GetRow(refid, pos)` as simply `GetRowsInRange(refid, pos, pos).front()`, and implement a binary search with the following sorting function on `fIndex`.

```
std::sort(fIndex.begin(), fIndex.end(), [](const IndexEntry &a, const
    IndexEntry &b) {
 if (a.refid != b.refid)
    return a.refid < b.refid;
 return a.pos < b.pos;
});
```

- **Eliminate redundant `fIndexMap`**

  The index currently maintains both a `vector<IndexEntry> fIndex` for serialization and a `map<std::pair<int32_t, int32_t>, int64_t>` for lookups, with `RebuildMap()` copying between them. However, as I noted in the previous issue, `fIndex` alone is fully capable of binary search by itself. `fIndexMap` and `RebuildMap()` become redundant and can be removed to reduce memory usage.

- **Configurable index granularity**

  The indexing intervals `kPositionInterval = 10000` and `kMappedInterval = 100` are hardcoded in `SamToNTuple.cxx`. I will make these configurable via function parameters and CLI flags. This

customization can help increase the speed for smaller or larger data samples, depending on the configured intervals.

- **Implement no-index query fallback**

  A quick inspection of legacy TTree `ramview_no_index.cxx` shows that if the data has no `fIndex`, it falls back to brute-force scanning, but RNTuple has no equivalent. I will implement a columnar scan fallback in `RAMNTupleView.cxx` that reads only the `refid`, `pos`, `flag`, and `cigar`. This extends the existing per-column view pattern already used in the codebase.

Task 5: Add support for Stat, View, Split, Merge, and Sort

RAMtools currently lacks essential functionalities that SAMtools provides out of the box. The following proposals are heavily inspired by the corresponding documentation on SAMtools.

- **Stats, IdxStats, FlagStat**

  Similar to the samtools stats, samtools idxstats, and samtools flagstat, I will create `tools/ramntuplestats.cxx`, `tools/ramntupleidxstat.cxx`, and `tools/ramntupleflagstat.cxx`.

  Given the 31 distinct metrics generated by `samtools stats`, developing its RAMtools equivalent will likely be the most time-intensive component of this task.

- **RAM→SAM export**

  As noted in GitHub issue #40, the existing `ConvertRAMNTupleToSAM()` in `RAMNTupleRecord.cxx` is bugged.

- **View**

  Inspired by samtools view and the existing TTree-only `tools/ramreader.cxx`, I will complete `tools/ramntupleview.cxx` with additional features such as viewing the first N records, random access by row number, region filtering, and selective column output.

- **Split, Merge**

  Currently, split is only supported by `samtoramntuple_split_by_chromosome()`, which only splits based on SAM files. Based on samtools split, I will create `tools/ramntuplesplit.cxx` that enables splitting on .root files as well.

  To complement the split feature, I will create `tools/ramntuplemerge.cxx` similar to samtools merge.

  Additionally, I noticed that `samtoramntuple_split_by_chromosome()` from `SamToNTuple.cxx` already sorts records within chromosomes by reading all SAM records into memory. Since `tools/ramntuplesplit.cxx` will operate on .root files where records are already stored in RNTuple, a different approach (e.g., index-based or streaming) will be needed to preserve sorted order within each chromosome. This can also help with developing the Sort functionality below.

- **Sort**

  Because genomic datasets are exceptionally large, in-memory sorting of entire records is infeasible. I will utilize the large-scale merge-sort algorithm's divide-and-conquer approach:

split into sorted chunks of temporary files, then merge, keeping $\mathcal{O}(N \log(N))$ time complexity. This function can be placed under `tools/ramntuplesort.cxx`.

Testing, Benchmarking, and Documenting

Development will be highly iterative rather than strictly sequential. While tasks #1 and #2 focus on establishing the benchmarking infrastructure, these benchmarks will be continuously updated alongside the feature development in tasks #3, #4, and #5.

For each new functionality or optimization introduced, I will add tests in `test/` for them and check for quantitative improvements through benchmarking, whether through the existing available benchmarking scripts or by creating new ones in `benchmark/`. For comparison, I will primarily compare my implementations against SAMtools. For the benchmarking dataset, I will use `GenerateSAMFile()` for test coverage to accelerate the testing process, and rely on real datasets collected in Task #1 to document final performance results.

In addition to documenting benchmark results, I will provide further documentation regarding the CLI usage for each new CLI flag introduced in the above tasks in a new folder `doc/`. This new folder is mostly consisted of `.ipynb`, `.md` files.

TIMELINE

Coding Period Begins	
Week 1 & 2 (May 25 – Jun 7)	Refactor the benchmark suite. Research quality bioinformatic datasets.
Week 3 & 4 (Jun 8 – Jun 21)	Load bioinformatic datasets. Test coverage and Documentation. **Deliverable:** Reproducible benchmark on heavy bioinformatics datasets.
Week 5 & 6 (Jun 22 – Jul 5)	Research various file formats. Decide cross-format comparison benchmark architecture.
Midterm Evaluations (Jul 6 – Jul 10)	
Week 7 & 8 (Jul 6 – Jul 19)	Load new file formats, run on existing datasets. Tests and Documentation. **Deliverable:** Cross-format benchmark suite comparing RNTuple against SAM, BAM, and CRAM.
Week 9 & 10 & 11 (Jul 20 – Aug 9)	Research and implement various compression algorithms and literature.
Week 12 & 13 (Aug 10 – Aug 23)	Benchmark compression algos. Test coverage and documentation. **Deliverable:** New data compression algorithms for genomic sequences.
Week 14 & 15 (Aug 24 – Sep 6)	Fix linear scan, eliminate redundant `fIndexMap`, configurable index granularity, and no-index query fallback.
Week 16 & 17 (Sep 7 – Sep 20)	Benchmark indexing & search speed. Test coverage and documentation. **Deliverable:** Optimize indexing and search capabilities for large genomic datasets.
Week 18 & 19 (Sep 21 – Oct 4)	Implement RAMNTuple Stats, IdxStats, FlagStat, and RAM $\rightarrow$ SAM export. Implement RAMNTuple View, Split, Merge, and Sort.
Week 20 & 21 (Oct 5 – Oct 18)	Benchmark new functionalities. Test coverage and documentation. **Deliverable:** Add support for Stat, View, Split, Merge, and Sort.
Week 22 & 23 (Oct 19 – Nov 1)	Buffer week. Catch up on any incomplete work. Address mentor feedback. Final tests, documentation, and final work submission.

Personal Background & Qualifications

Hi! I am Jeffrey Zhang, GitHub username RedBlueBird, a 3rd-year Physics undergraduate at Nagoya University, Japan. I have been programming since elementary school and have developed a passion for coding. I created video games, participated in hackathons, competed in competitive programming in C++, like the United States of America Computing Olympiad Gold Division 75th rank in 2023, and also had a software engineering internship at Rakuten, known for its data-intensive e-commerce platform in Japan. More details about me are available on my website: https://redbluebird.github.io/.

Here are some previous efforts I have made to the `compiler-research/ramtools` repository.

- [Merged PR #39] Add Test Coverage for IndexGetRowsInRange
- [Merged PR #43] Add Test Coverage for Getters in FRAMNTupleRecord.cxx
- [Merged PR #45] Fix DecodeSequence bug
- [Closed Issue #44] Bug: GetSEQ, DecodeSequence not working as intended
- [Draft PR #48] Chore clang-tidy warnings for ramcoretests.cxx