Using ROOT in the field of genome sequencing

Aditya Pandey

April 1, 2025

Mentoring Organization:- CERN-HSF

Mentors:-

Name :- Martin Vassilev
Email:- **mvassilev@uni-plovdiv.bg**

Name :- Jonas Rembser
Email:- **Jonas.Rembser@cern.ch**

Name :- Fons Rademakers
Email:- **Fons.Rademakers@cern.ch**

Abstract

This project aims to advance genomic data management by implementing ROOT's next-generation RNTuple format for sequence alignment storage. Beginning with validation of previous GeneROOT benchmarks showing 4x performance gains with TTree, we will then extend these capabilities with RNTuple technology. Genomic sequencing data volumes are growing exponentially, creating performance bottlenecks in traditional formats. RNTuple's improved memory mapping, type safety through templated interfaces, and parallelization capabilities position it as an ideal solution. We will systematically compare compression algorithms, implement file splitting strategies, and benchmark against established formats. The project will deliver optimized tools for handling rapidly growing genomic datasets, potentially establishing a new standard for high-performance genomic data analysis.

Contents

1. About Me

1.1 Personal Details

- **Name:** Aditya Pandey
- **Email:** adityapand3y666@gmail.com
- **GitHub:** AdityaPandeyCN
- **Location:** Ranchi, India
- **Blog:** adityapandeycn.github.io/personalblog/

1.2 Motivation

My experience with C++ programming and low-level optimization makes me well-suited for this project. The skills I developed during my previous GSoC, particularly in validation, error handling, and creating simulation-ready models, will help me in my work with GeneROOT. My Rubik's cube solver project demonstrates my ability to implement practical solutions to complex problems, showcasing my technical creativity and problem-solving approach.

1.3 Time Availability

I am fully committed to dedicating 60 hours per week to this project during the initial GSoC period, leveraging my complete availability during summer break from May to August. After August, I will maintain a consistent 40-hour weekly commitment, with no other prior commitments.

1.4 Technical Profile

Programming and Development Experience

I have over 2 years of programming experience particularly relevant to this project:

Languages and Technologies

- **Languages:** C++, Python, C, Rust, JavaScript
- **Libraries/Frameworks:** Django, React.js
- **Optimization Techniques:** Algorithmic optimization

Relevant Experience

- Google Summer of Code with INCF: Developed an interactive Python tool with robust error handling and validation checks for neuronal morphology conversion

  My GSoC 2024 work

- Strong problem-solving skills: Solved 150+ problems on LeetCode
- Qualified for the Pre-Regional Mathematics Olympiad(1st stage to IMO)

2. Description

This project extends GeneROOT by implementing ROOT's next-generation RNTuple format for genomic data storage and analysis. It begins by reproducing previous benchmark results, then develops an RNTuple-based system to replace the TTree implementation used in earlier GeneROOT work. The project will create conversion utilities between standard genomic formats (BAM/CRAM) and RNTuple, benchmark different compression algorithms, explore file splitting strategies, and document performance comparisons.

3. Implementation Plan

The project will be executed in two distinct stages:

3.1 Overview

Stage 1: Reproduction and Baseline Establishment

The first stage focuses on reproducing previous GeneROOT results and establishing the foundation for RNTuple implementation:

1. Set up the development environment with ROOT, htslib, and necessary genomic tools
2. Reproduce the TTree-based ROOT Alignment Maps (RAM) implementation from previous GeneROOT work
3. Verify and document performance comparisons between BAM/CRAM formats and the TTree implementation
4. Investigate and compare Samtools' latest BAM/CRAM compression strategies with the ROOT Alignment Maps (RAM)
5. Benchmark different compression strategies (ZLIB, LZMA, LZ4) with the existing TTree format
6. Develop and test the basic genomic data model design that will later be implemented with RNTuple

Stage 2: RNTuple Implementation and Advanced Features

The second stage involves implementing the RNTuple storage backend and developing advanced features:

1. Implement a genomic data model using RNTuple's templated field system
2. Develop conversion utilities between BAM/CRAM formats and the RNTuple implementation
3. Implement and test different file splitting strategies (by chromosome, region, read group)
4. Create efficient query tools for genomic region retrieval

5. Conduct comprehensive benchmarking comparing RNTuple performance against both TTree and traditional formats
6. Produce final comparison report with performance analysis and recommendations

3.2 Detailed Description

I plan to execute the conversion part in 2 stages.

Stage 1: Reproduction and Baseline Establishment

High-Level Goal

- To reproduce and validate existing GeneROOT results with the TTree format
- To establish reliable baseline metrics for comparing future RNTuple implementations
- To identify and address specific performance bottlenecks in the current implementation
- To optimize the existing codebase before transitioning to the new format

Workflow

Completed Work

I have already made significant progress on reproducing and optimizing the existing GeneROOT implementation:

Environment and Dataset Preparation:

- Successfully set up the development environment with ROOT and htslib
- Discovered and leveraged existing tools in the ramtools repository
- Acquired test files of varying sizes for comprehensive benchmarking:

    - 9GB BAM file (Pond-154926.bam)
    - 15GB BAM file (HG00154.mapped.ILLUMINA.bwa.GBR.low_coverage.20101123.bam)
- Created ROOT files with different compression algorithms (ZLIB, LZ4, LZMA)
- Created a PR to the GeneROOT repository with optimized ramview implementation

Key Code Optimizations Implemented

High-Level Goal

My optimizations target critical performance bottlenecks in genomic data processing, focusing on improving query speed while maintaining memory efficiency for large datasets.

An overview of the changes in ramview

Advanced Search Algorithm

Problem: The original GeneROOT implementation uses a linear search to find genomic positions, which becomes extremely inefficient for large files where the target region might be millions of entries away from the starting point. This results in unnecessary I/O operations and slow queries, especially when users want to examine distant genomic regions.

Solution: I replaced the linear search with a two-phase approach combining exponential and binary search algorithms. This dramatically reduces the number of entries that need to be examined to find a specific genomic region:

```
// ORIGINAL: Simple linear search through all entries
for (; start_entry < end_entry; start_entry++) {
   t->GetEntry(start_entry);
   if (r->GetPOS() + r->GetSEQLEN() > range_start) {
      break;
   }
}

// OPTIMIZED: Exponential search followed by binary search
Long64_t jump = 1;
Long64_t curr = start_entry;
while (curr < end_entry) {
   t->GetEntry(curr);
   if (r->GetPOS() + r->GetSEQLEN() > range_start) {
      found_start = true;
      break;
   }
   start_entry = curr;
```

```
    jump *= 2;
    curr = std::min(curr + jump, end_entry - 1);
}
```

Batch Processing: Problem: The original implementation processes genomic entries one at a time, requiring a separate disk I/O operation for each entry. For large genomic regions containing thousands or millions of entries, this creates a severe I/O bottleneck, with most processing time spent waiting for disk access rather than analyzing data.

Solution: I implemented dynamic batch processing that loads multiple entries in a single operation, drastically reducing I/O overhead:

```
// NEW: Adaptive batch sizing function
inline int getOptimalBatchSize(Long64_t regionSize) {
    if (regionSize < 100) return 20;
    if (regionSize < 1000) return 100;
    if (regionSize < 10000) return 1000;
    if (regionSize < 100000) return 5000;
    return 10000;
}
// Batch processing implementation
const int BATCH_SIZE = getOptimalBatchSize(end_entry - start_entry);
std::vector<Long64_t> entryBuffer(BATCH_SIZE);
// Process entries in optimally-sized batches
while (j < end_entry) {
    // Fill the buffer with next batch of entries
    int batchEntries = 0;
    for (int i = 0; i < BATCH_SIZE && j < end_entry; i++, j++) {
        entryBuffer[i] = j;
        batchEntries++;
    }
    // Process the entire batch at once
    for (int i = 0; i < batchEntries; i++) {
        t->GetEntry(entryBuffer[i]);
        // Process entry...
    }
}
```

Resource Optimization

Problem: The original code uses fixed memory allocations regardless of file size, which is inefficient for both small and large datasets. For small files, this wastes memory; for large genome files (often 10-30GB), it causes excessive paging and poor performance.

Solution: I implemented dynamic resource allocation that scales based on file size.

```
// NEW: File-size based optimizations
```

```
Long64_t fileSize = f->GetSize();
if (fileSize > 1024*1024*1024) { // 1 GB
   t->SetMaxVirtualSize(256*1024*1024); // 256 MB for large files
}
```

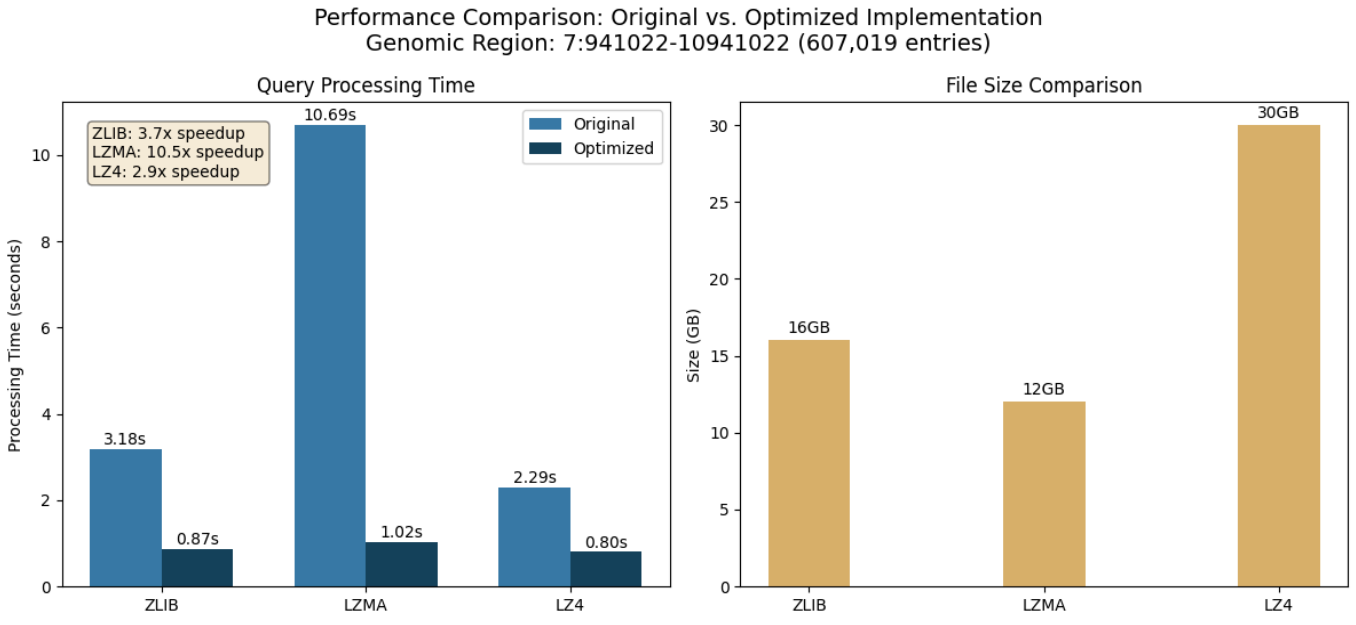
Selective Branch Management:

Problem: Genomic data is complex with many fields, but most queries only need a small subset of this data. The original implementation loads all data branches even when only position information is needed, wasting memory and I/O bandwidth.

Solution: I implemented selective branch loading to focus resources on only the necessary data.

```
// ORIGINAL: Basic branch enabling
if (b->GetSplitLevel() > 0)
   t->SetBranchStatus("RAMRecord.*", 0);

// OPTIMIZED: More control
t->SetBranchStatus("*", 0); // Disable all branches
t->SetBranchStatus("RAMRecord.v_refid", 1);
t->SetBranchStatus("RAMRecord.v_pos", 1);
t->SetBranchStatus("RAMRecord.v_lseq", 1);
```

Result of benchmarking region view of RAM files



Benchmark performed on 15GB BAM file with optimized ramview.C implementation

Note:- All the scripts used for benchmarking can be found here

Remaining Work for Stage 1

High-Level Goal

- To comprehensively test the optimized TTree implementation across various file sizes and formats
- Investigate and compare the latest compression strategies used by Samtools for conversions to BAM, with RAM (ROOT Alignment Maps)
- To establish standardized benchmark protocols for consistent performance measurement
- To document performance patterns and scaling behaviors to inform RNTuple implementation
- To identify specific areas where RNTuple could provide the greatest performance improvements

1. Analysis of Latest Compression Strategies used by Samtools

High-Level Goal

- Evaluate modern compression techniques from Samtools/HTSlib for genomic data.
- Integrate the most efficient methods into ROOT's storage system for optimal space and speed.
- Enable flexible compression choices to suit different genomic workflows and data sizes.

Analysis Strategy

1.1 Comprehensive Analysis of Current Compression Technologies

I will conduct a thorough examination of Samtools/HTSlib compression implementations, focusing on:

BGZF (Blocked GZIP Format) in BAM Files

- Block structure analysis: Study the 64KB block architecture that enables random access while maintaining gzip compatibility
- Parameter tuning: Investigate the nine compression levels (1-9) and their impact on genomic data
- Multi-threading implementation: Analyze how `bgzf_mt` API implements parallel compression/decompression

CRAM Advanced Codecs

- rANS (Asymmetric Numeral Systems): Study both 4×8 (CRAM 2.1/3.0) and 4×16 implementations
- External codecs: Analyze bzip2 and LZMA integration techniques
- CRAM 3.1 transforms: Examine advanced techniques like Nx16 interleaving, RLE, bit-packing, and striped rANS

1.2 Implementation and Testing Methodology

My approach to comparing these with ROOT's compression capabilities will include:

- Codec library integration: Evaluate feasibility of porting or interfacing with HTSlib's compression libraries
- ROOT-native implementation: Develop ROOT-compatible versions of key algorithms where direct integration isn't possible
- Reference-based compression: Implement specialized reference sequence handling similar to CRAM
- Adaptive selection: Create a dynamic system to select optimal compression methods based on data characteristics

2. Comprehensive File Format and Size Testing

- Acquire and test additional files of varying sizes following My Mentor’s guidance:
    - Small (1-5GB), Medium (5-15GB), Large (15-30GB), Very Large (30GB+)
    - Test across all formats: BAM, CRAM, FASTA, FASTQ
- Process the 15GB BAM file(HG00154.mapped.ILLUMINA.bwa.GBR.low_coverage.20101123.bam) already downloaded
- Download additional large datasets (preferably 20GB+ files)
- Create corresponding ROOT files with all compression methods

3. Complete Benchmark Matrix

- Create a comprehensive benchmark matrix:
    - File Formats: BAM vs CRAM vs ROOT (ZLIB/LZMA/LZ4)
    - File Sizes: Small vs Medium vs Large vs Very Large
    - Operations: Region queries, whole file processing, conversion times
    - Genomic Regions: Cover some complex regions to solidify the benchmarking
- Document performance patterns as file size increases
- Identify scaling factors and potential bottlenecks

4. RAM Implementation Analysis

- Analyze RAM data model for compatibility with RNTuple
- Evaluate TTree performance characteristics at different file sizes
- Identify performance patterns that could inform RNTuple implementation

5. Documentation and Analysis

- Create detailed performance graphs showing scaling behavior
- Document all benchmark results with statistical significance
- Prepare comprehensive report comparing results with the original GeneROOT benchmarks
- Include solid conclusions about performance based on larger file testing

Transition Preparation for Stage 2

High-Level Goal

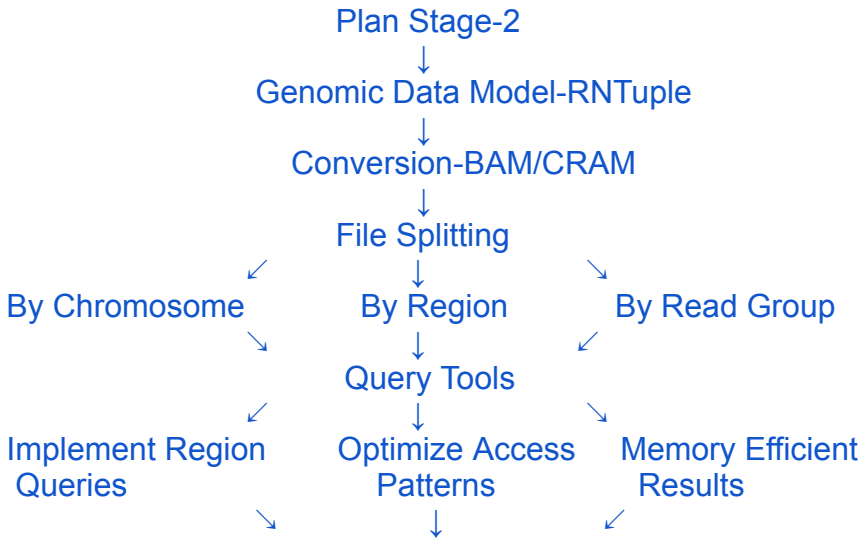
- To analyze the optimized TTree codebase for insights applicable to RNTuple design
- To identify critical data access patterns and performance hotspots that affect genomic data processing
- To design adapter interfaces that will facilitate migration between implementations
- To create a prototype RNTuple data model based on TTree implementation insights

Stage 2: RNTuple Implementation for Genomic Data

High-Level Goal

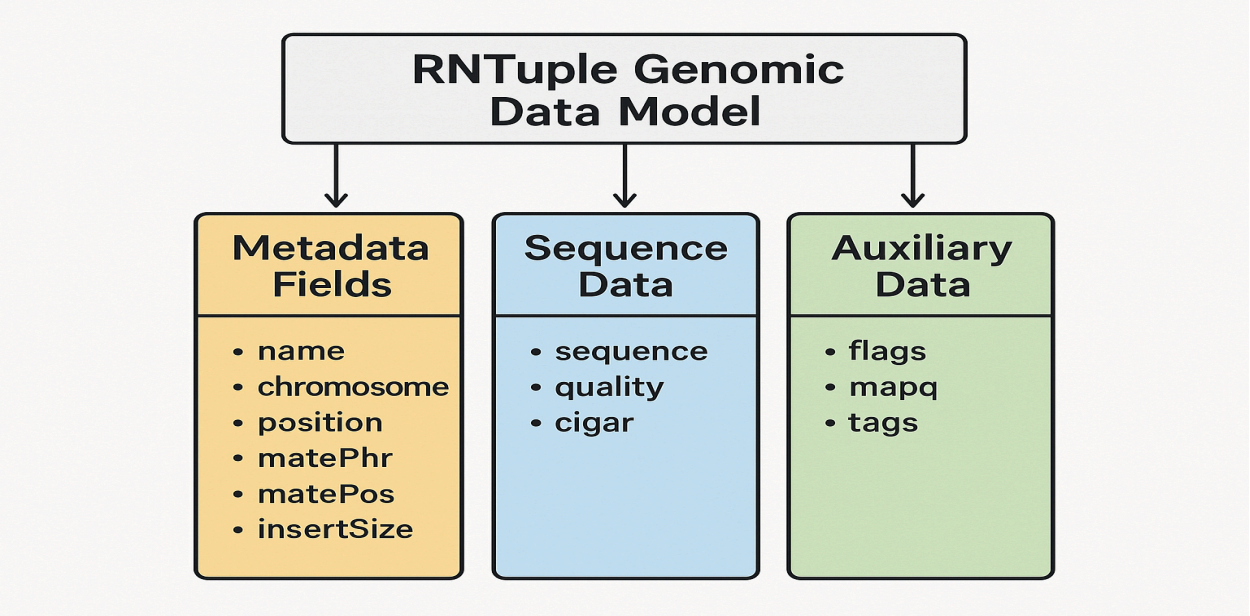
- To create an optimized representation of genomic alignment data balancing completeness with performance
- To leverage RNTuple's type-safe field system to prevent common programming errors
- To organize genomic information in a columnar structure enabling efficient memory access
- To design a model that naturally accommodates the hierarchical nature of genomic data

Workflow



Benchmarking
↓
Final Report

1. Data Model Design



We plan to create a specialized RNTuple data model that maps genomic alignment data (from BAM/CRAM formats) to RNTuple's type-safe field system. Each alignment record contains essential genomic information in a type-safe, columnar structure.

Code Snippet

```
// Define RNTuple model for genomic alignment data
std::shared_ptr<RNTupleModel> CreateAlignmentModel() {
    auto model = RNTupleModel::Create();

    // Core alignment fields
    model->MakeField<std::string>("name");       // Read name
    model->MakeField<std::string>("chromosome"); // Reference name
    model->MakeField<int32_t>("position");       // Alignment position
    model->MakeField<uint16_t>("flags");         // BAM flags
    model->MakeField<uint8_t>("mapq");           // Mapping quality
    model->MakeField<std::string>("cigar");      // CIGAR string
    model->MakeField<std::string>("sequence");   // Read sequence
    model->MakeField<std::string>("quality");    // Base qualities

    // Tags stored as key-value pairs
    model->MakeField<std::vector<std::pair<std::string, std::string>>>("tags");

    return model;
}
```

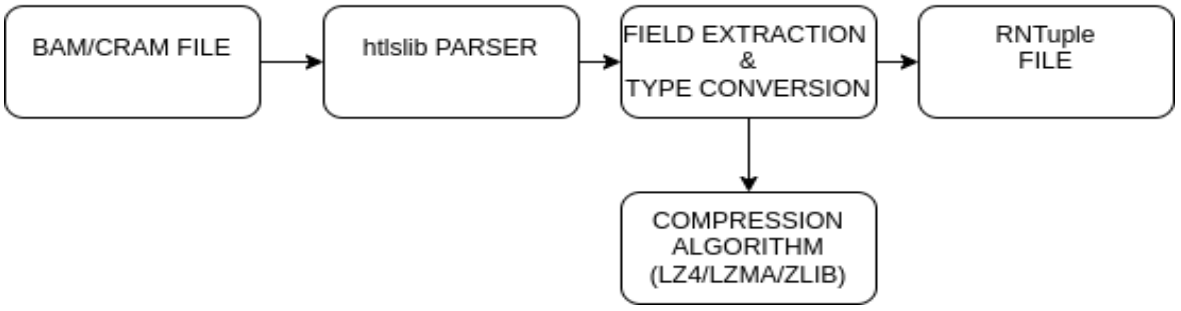
This model leverages RNTuple's templated interfaces to provide compile-time type checking, preventing common errors in genomic data processing. The columnar structure allows efficient

memory access patterns for specific fields, which is particularly valuable for genomic queries that often access only chromosome and position information.

2. Optimized Conversion Pipeline

High-Level Goal

- To efficiently transform standard genomic formats (BAM/CRAM) into RNTuple storage
- To handle massive data volumes (10-30GB per sample) with minimal computational overhead
- To provide configurable compression options balancing file size against processing speed
- To implement memory-efficient batch processing for large genomic datasets



We plan to develop efficient converters between standard genomic formats (BAM/CRAM) and RNTuple storage. The implementation uses htslib for parsing input files and incorporates batched processing and memory management techniques to handle massive genomic datasets (often 10-30GB per sample).

```
// Core conversion function (simplified)
int ConvertToRNTuple(const char* inputFile, const char* outputFile) {
    // Open input CRAM/BAM file
    samFile* in = sam_open(inputFile, "r");
    bam_hdr_t* header = sam_hdr_read(in);
    // Create RNTuple model and writer
    auto model = CreateAlignmentModel();
    auto ntuple = RNTupleWriter::Recreate(std::move(model), "Alignments",
outputFile);
    // Get field handles
    auto name =
ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("name");
    auto chromosome =
ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("chromosome");
    auto position =
ntuple->GetModel()->GetDefaultEntry()->Get<int32_t>("position");
```

```
    // ... get other fields ...
    // Process alignments
    bam1_t* aln = bam_init1();
    while (sam_read1(in, header, aln) >= 0) {
        // Extract data from BAM record
        *name = bam_get_qname(aln);
        *chromosome = header->target_name[aln->core.tid];
        *position = aln->core.pos + 1;
        // ... extract other fields ...
        // Write to RNTuple
        ntuple->Fill();
    }
    // Cleanup
    bam_destroy1(aln);
    bam_hdr_destroy(header);
    sam_close(in);

    return 0;
}
```

The converter supports various compression algorithms (ZLIB, LZMA, LZ4, ZSTD) with configurable settings, allowing users to balance file size against processing speed:

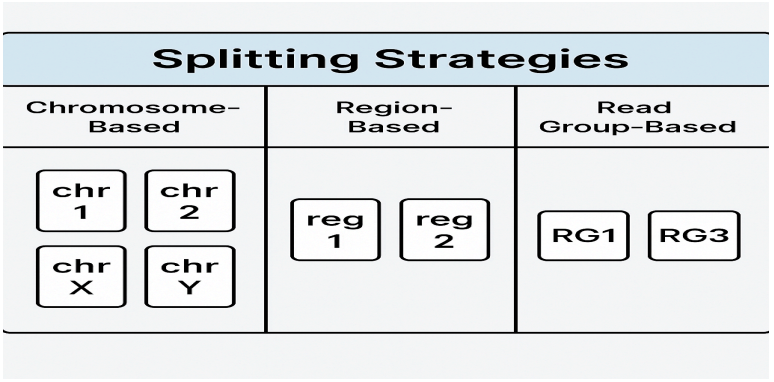
Code Snippet

```
// Configure RNTuple compression options
RNTupleWriter::RNTupleWriteOptions ConfigureCompression(const std::string& algorithm) {
    RNTupleWriter::RNTupleWriteOptions options;

    if (algorithm == "zlib") {
        options.SetCompression(ROOT::RCompressionSetting::EAlgorithm::kZLIB, 6);
    } else if (algorithm == "lzma") {
        options.SetCompression(ROOT::RCompressionSetting::EAlgorithm::kLZMA, 9);
    } else if (algorithm == "lz4") {
        options.SetCompression(ROOT::RCompressionSetting::EAlgorithm::kLZ4, 4);
    } else if (algorithm == "zstd") {
        options.SetCompression(ROOT::RCompressionSetting::EAlgorithm::kZSTD, 5);
    }

    // Configure optimal cluster size for genomic data
    options.SetNElements(64000);

    return options;
}
```

3. **Advanced Splitting Strategies**

High-Level Goal

- To optimize both storage organization and query performance through intelligent data partitioning
- To address different access patterns common in genomic research (chromosome-level, region-specific)
- To enable parallel processing opportunities through logical data separation
- To reduce I/O overhead when analyzing specific genomic regions



We plan to implement multiple file organization strategies to optimize storage and query performance. The chromosome-based splitting creates separate files for each chromosome, enabling parallel processing and reducing I/O when analyzing specific genomic regions:

Code Snippet

```
int SplitByChr(const char* inputFile, const char* outputPrefix) {
    // Open input file
    samFile* in = sam_open(inputFile, "r");
    if (!in) return -1;

    bam_hdr_t* header = sam_hdr_read(in);
    if (!header) {
        sam_close(in);
        return -1;
    }
    // Map of chromosome to RNTuple writers
    std::map<std::string, std::shared_ptr<RNTupleWriter>> writers;
    // Process alignments
    bam1_t* aln = bam_init1();
    while (sam_read1(in, header, aln) >= 0) {
        // Skip unmapped reads
        if (aln->core.tid < 0) continue;
        std::string chr = header->target_name[aln->core.tid];
        // Create new writer for this chromosome if needed
        if (writers.find(chr) == writers.end()) {
            auto model = CreateAlignmentModel();
            std::string filename = std::string(outputPrefix) + "/" + chr + ".root";
            writers[chr] = RNTupleWriter::Recreate(std::move(model), "Alignments",
filename);
        }
```

```
        // Get the writer for this chromosome
        auto& ntuple = writers[chr];
        // Fill data fields
        *ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("name") =
bam_get_qname(aln);
        *ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("chromosome") = chr;
        *ntuple->GetModel()->GetDefaultEntry()->Get<int32_t>("position") = aln->core.pos +
1;
        *ntuple->GetModel()->GetDefaultEntry()->Get<uint16_t>("flags") = aln->core.flag;
        *ntuple->GetModel()->GetDefaultEntry()->Get<uint8_t>("mapq") = aln->core.qual;
        // Extract and store sequence
        int l_seq = aln->core.l_qseq;
        std::string sequence;
        sequence.reserve(l_seq);
        uint8_t* seq = bam_get_seq(aln);
        for (int i = 0; i < l_seq; i++) {
            sequence.push_back("=ACMGRSVTWYHKDBN"[bam_seqi(seq, i)]);
        }
        *ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("sequence") = sequence;
        // Extract quality scores
        std::string quality;
        quality.reserve(l_seq);
        uint8_t* qual = bam_get_qual(aln);
        for (int i = 0; i < l_seq; i++) {
            quality.push_back(33 + qual[i]);
        }
        *ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("quality") = quality;
        // Extract CIGAR string
        uint32_t* cigar = bam_get_cigar(aln);
        std::string cigarStr;
        for (uint32_t i = 0; i < aln->core.n_cigar; ++i) {
            cigarStr += std::to_string(bam_cigar_oplen(cigar[i]));
            cigarStr += bam_cigar_opchr(cigar[i]);
        }
        *ntuple->GetModel()->GetDefaultEntry()->Get<std::string>("cigar") = cigarStr;
        // Write record
        ntuple->Fill();
    }
    // Cleanup
    bam_destroy1(aln);
    bam_hdr_destroy(header);
    sam_close(in);
    return 0;
}
```

Beyond chromosome-based splitting, we'll also implement:

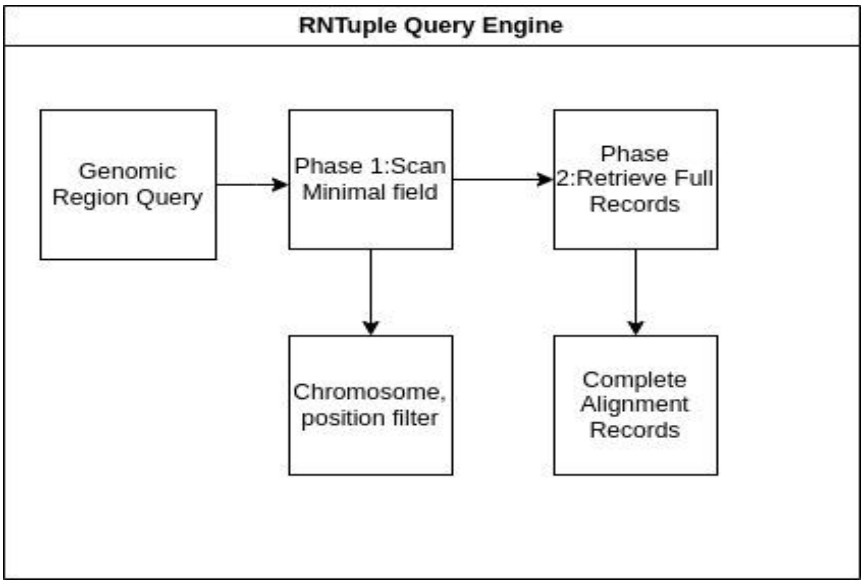
- **Region-based splitting:** Divides the genome into configurable regions (fixed-size or feature-based)
- **Read group splitting:** Separates multi-sample datasets, particularly useful in population genomics studies

4. High-Performance Query Engine

High-Level Goal

- To fully leverage RNTuple's columnar structure for rapid retrieval of genomic regions
- To implement a two-phase query approach minimizing unnecessary data access
- To provide memory-efficient handling of large result sets common in genomic analysis
- To enable complex filtering based on alignment properties without performance penalties

This two-phase query approach (first scanning minimal fields, then retrieving full records) significantly accelerates genomic region queries by reducing the amount of data accessed. The implementation also provides memory-efficient handling of large result sets and support for filtering based on alignment properties.



Code Snippet

```
// Query alignments in a genomic region (simplified)
std::vector<AlignmentRecord> QueryRegion(const char* ntuplePath, const char* regionStr) {
    std::vector<AlignmentRecord> results;
    // Parse region (e.g., "chr1:1000-2000")
    std::string region = regionStr;
    size_t colonPos = region.find(":");
    std::string targetChr = region.substr(0, colonPos);

    size_t dashPos = region.find("-", colonPos);
    int32_t targetStart = std::stoi(region.substr(colonPos + 1, dashPos - colonPos - 1));
    int32_t targetEnd = std::stoi(region.substr(dashPos + 1));
    // Open RNTuple file
    auto ntuple = RNTupleReader::Open("Alignments", ntuplePath);

    // Efficient query using minimal fields first
    auto view = ntuple->GetView<std::string, int32_t>({"chromosome", "position"});
```

```
    // Process matching entries
    for (auto i : ntuple->GetEntryRange()) {
        auto [chr, pos] = view(i);

        if (chr == targetChr && pos >= targetStart && pos <= targetEnd) {
            // Add to results (full record would be retrieved here)
            AlignmentRecord rec;
            // ... populate record fields ...
            results.push_back(rec);
        }
    }
    return results;
}
```

Note on Code Samples: The code snippets included throughout this proposal are provided to demonstrate technical understanding of the implementation approach. These are representative examples only and will be refined and expanded in collaboration with my mentor during the project development. The final implementation will incorporate best practices and optimizations based on mentor feedback and initial testing.

5. Comprehensive Benchmarking

We plan to create a benchmarking framework to compare performance across formats (BAM, CRAM, TTree-RAM, RNTuple-RAM) using standardized tests:

Region query performance with varying region sizes
File size comparisons with different compression strategies
Memory consumption patterns
Scaling behavior with increasing file sizes

Expected Benefits

Improved Query Performance: RNTuple's columnar structure accelerates genomic region queries.
Enhanced Type Safety: RNTuple's templated interfaces provide compile-time type checking.
Better Memory Efficiency: More efficient memory mapping reduces RAM requirements.
Optimized Storage: Specialized compression and splitting strategies balance file size with query performance.
Future-Proof Architecture: RNTuple represents ROOT's next-generation storage format

4. Timeline for GeneROOT Project: Stage 1 and Stage 2 Implementation

Time Frame	Start Date	End Date	Task	Deliverable
Week 0: Community Bonding	May 8, 2025	May 14, 2025	• Study GeneROOT codebase • Set up development environment • Initial project planning	Project roadmap and development setup
Week 1: Community Bonding	May 15, 2025	June 1, 2025	• Detailed project architecture review • Mentor consultations • Finalize implementation strategy	Comprehensive project strategy document
Week 1 (Stage 1)	June 2, 2025	June 8, 2025	• Begin reproducing TTree implementation • Initial optimization of ramview.C • Start benchmarks with 9GB BAM file	Initial benchmark scripts and performance baseline
Week 2 (Stage 1)	June 9, 2025	June 15, 2025	• Complete TTree implementation reproduction • Finalize ramview.C optimizations • Complete 9GB BAM file benchmarks	Optimized TTree implementation with benchmark results
Week 3 (Stage 1)	June 16, 2025	June 22, 2025	• Expand benchmarking to 15GB+ files • Test ZLIB compression algorithm • Initial performance documentation	Comprehensive benchmarks with ZLIB compression
Week 4 (Stage 1)	June 23, 2025	June 29, 2025	• Test LZ4 and LZMA compression • Compare compression performance • Document comparative results	Comparative compression algorithm performance report

Buffer Week 1	**June 30, 2025**	**July 6, 2025**	**• Catch up on any delayed Stage 1 tasks • Additional testing if needed • Further optimization of problematic areas**	**Refined Stage 1 implementation and stable codebase**
Week 5 (Transition)	July 7, 2025	July 13, 2025	• Complete Stage 1 documentation • Begin RNTuple data model implementation • Create initial converter prototype	Stage 1 comprehensive documentation
Week 6 (Transition)	July 14, 2025	July 20, 2025	• Refine RNTuple data model • Develop converter prototype • Prepare midterm deliverables	RNTuple converter prototype
Midterm Evaluation	July 21, 2025	July 25, 2025	• Submit midterm evaluation • Review progress with mentors • Finalize RNTuple approach	Midterm project status and refined implementation plan
Week 7 (Stage 2)	July 26, 2025	August 1, 2025	• Develop optimized RNTuple converter • Implement multiple compression options • Start chromosome-based splitting	Optimized RNTuple converter with compression support
Week 8 (Stage 2)	August 2, 2025	August 8, 2025	• Complete chromosome-based splitting • Refine converter performance • Preliminary RNTuple benchmarks	Chromosome-based splitting implementation
Week 9 (Stage 2)	August 9, 2025	August 15, 2025	• Implement region-based splitting • Develop initial query tools • Start RNTuple benchmarks	Region-based splitting and query tool prototype

Buffer Week 2	**August 16, 2025**	**August 22, 2025**	**• Address remaining implementation issues • Additional optimization if needed • Complete any delayed Stage 2 tasks**	**Stabilized implementation with optimized performance**
Week 10 (Stage 2)	August 23, 2025	August 29, 2025	• Complete query tools • Comprehensive RNTuple benchmarks • Performance optimization	Completed query tools with performance benchmarks
Week 11 (Stage 2)	August 30, 2025	August 31, 2025	• Comparative benchmarking (TTree vs RNTuple) • Detailed performance analysis • Finalize documentation	Comparative performance analysis
Final Submission	September 1, 2025	September 1, 2025	• Submit final code • Submit benchmark report • Submit implementation documentation	Final project submission package

Note: Stage 1 focuses on reproducing and optimizing the existing TTree implementation while Stage 2 develops the new RNTuple implementation. The timeline includes continuous testing throughout both

5. Contributions and Communications

5.1 Communication with Mentors

I will keep updating my mentors with weekly emails and communicate through open source chat channels. I will also try to connect over video meetings with my mentors when they are available and I am confused with some aspects of the projects.

5.2 Contributions

I plan to make meaningful contributions to GeneROOT by:

- Completing the reproduction and analysis of existing TTree-based benchmarks with larger datasets
- Implementing an efficient RNTuple data model for genomic alignment data
- Developing high-performance conversion utilities between BAM/CRAM and RNTuple formats

- Creating multiple splitting strategies (chromosome, region, read group) to optimize access patterns
- Implementing optimized query tools leveraging RNTuple's columnar structure
- Conducting comprehensive benchmarking to quantify performance improvements
- Providing detailed documentation and analysis to guide future development

My preliminary work optimizing ramview.C has already demonstrated substantial performance improvements, and I'm excited to extend these gains through the implementation of RNTuple for genomic data storage.

Pull Requests:

1. Raised PR to the GeneRoot Repo with work on Ramview (**Open)**

6. References

- https://indico.cern.ch/event/655464/
- https://root.cern/
- https://github.com/GeneROOT
- https://root.cern/doc/v622/md_tree_ntuple_v7_doc_README.html
- https://www.htslib.org/
- https://www.internationalgenome.org/data(FOR BAM AND CRAM FILES)