# CS4545/CS6545 Project Report: Coded Blockchain Based Range Queries

Brahmpreet Singh
Faculty of Computer Science
University of New Brunswick
bsingh2@unb.ca

Dineth Mudugamuwa Hewage
Faculty of Computer Science
University of New Brunswick
dineth.m@unb.ca

Nicholas George Allison
Faculty of Computer Science
University of New Brunswick
nalliso1@unb.ca

## ABSTRACT

This project introduces a novel approach to blockchain data storage and retrieval through the implementation of coded blockchain technology for efficient range queries on historical data. By utilizing error correction codes, our system fragments blockchain blocks across multiple distributed storage nodes while maintaining data integrity and fault tolerance. We developed an index structure based on B+ trees that enables efficient range queries without requiring nodes to store complete blockchain data. Each node stores only 60% of the original block data, achieving a 40% redundancy while retaining the ability to reconstruct complete blocks. The system provides a practical solution for resource-constrained blockchain applications that require efficient historical data access.

## KEYWORDS

Blockchain; Error Correction Codes; Distributed Storage; Range Queries; B+ Tree; Data Fragmentation; Fault Tolerance; Coded Storage

## 1. INTRODUCTION

Blockchain technology has emerged as a transformative approach for maintaining immutable, decentralized ledgers of transactions across various domains. However, as blockchain networks grow, the storage requirements for maintaining a complete copy of the chain become prohibitive for many applications, particularly on resource-constrained devices. Additionally, efficiently querying historical blockchain data remains challenging, especially when specific data ranges are required.

This project addresses these challenges by implementing a coded blockchain system that distributes encoded fragments of blocks across multiple storage nodes while maintaining the ability to recover complete blocks even when some fragments are unavailable. We use the zfec library to apply Reed-Solomon encoding and employ deterministic hashing to evenly distribute fragments across storage nodes without requiring centralized coordination. Furthermore, we develop an efficient indexing mechanism that enables range queries on historical blockchain data without requiring access to the entire chain. By combining coding theory with advanced indexing techniques, our solution offers a practical approach to maintaining blockchain integrity while reducing storage requirements and improving query efficiency.

## 2. RELATED WORKS

Several approaches have been proposed to address the storage and query efficiency challenges in blockchain systems:

1. Sharding Techniques: Blockchain sharding divides the network into smaller partitions to process transactions in parallel, as seen in Ethereum 2.0 and Zilliqa. While these approaches improve transaction throughput, they don't directly address historical data storage efficiency.
2. Pruning Methods: Bitcoin and Ethereum implement pruning to remove spent transaction outputs, but this can limit historical query capabilities.

3. Authenticated Data Structures: Works by Miller et al. and Reyzin et al. have explored authenticated data structures for efficient blockchain querying, but these typically require nodes to maintain complete indices.
4. Erasure Coding in Distributed Systems: Systems like Hadoop HDFS and Ceph use erasure coding to reduce storage overhead while maintaining fault tolerance, inspiring our approach to blockchain storage.
5. Range Query Optimization: Traditional database techniques like B+ trees and skip lists have been adapted for blockchain contexts by researchers like Xu et al., but these are rarely combined with storage efficiency techniques.

Our work differs from these approaches by uniquely combining error correction coding for storage efficiency with indexing structures for range query support, addressing both challenges simultaneously.

## 3. PROBLEM STATEMENT

The core challenges our project addresses are:

**1. Storage Overhead: Traditional blockchain implementations require each node to store the entire blockchain, which becomes prohibitively expensive as the chain grows.**
**1. Query Inefficiency: Retrieving historical data based on value ranges (e.g., finding transactions within a specific value range) typically requires scanning the entire blockchain, resulting in poor performance.**
**1. Resource Constraints: Many potential blockchain applications run on devices with limited storage and processing capabilities, making full blockchain participation impractical.**
**1. Fault Tolerance: Any solution must maintain blockchain's inherent fault tolerance, ensuring data availability even when some nodes are unavailable.**

We aim to develop a system that reduces per-node storage requirements while maintaining data integrity and enabling efficient range queries on historical blockchain data. The solution should be robust against node failures, ensuring that blockchain data remains recoverable when a threshold number of nodes are available.

## 4. OUR APPROACH

### *4.1 System Architecture*

Our coded blockchain system consists of four core components:

- *Blockchain Core: Implements the fundamental blockchain structure with blocks containing transaction data and cryptographic links.*
- *Coding Layer: Applies error correction codes to fragment blocks across multiple storage nodes while ensuring recoverability.*
- *Distributed Storage: Manages the distribution and retrieval of block fragments across a network of nodes.*
- *Index Manager: Maintains efficient B+ tree indices for range query support on various data attributes.*

### *4.2 Error Correction Coding*

We implement a systematic approach to block fragmentation using the zfec library, which implements Reed-Solomon encoding. Each block is encoded into n fragments such that any k fragments (where $k < n$) are sufficient to reconstruct the original block. This provides a (n-k)/n redundancy that ensures fault tolerance while reducing the storage requirement at each node to 1/k of the original data size.

The encoding process works as follows:
1. A block is serialized into a byte representation

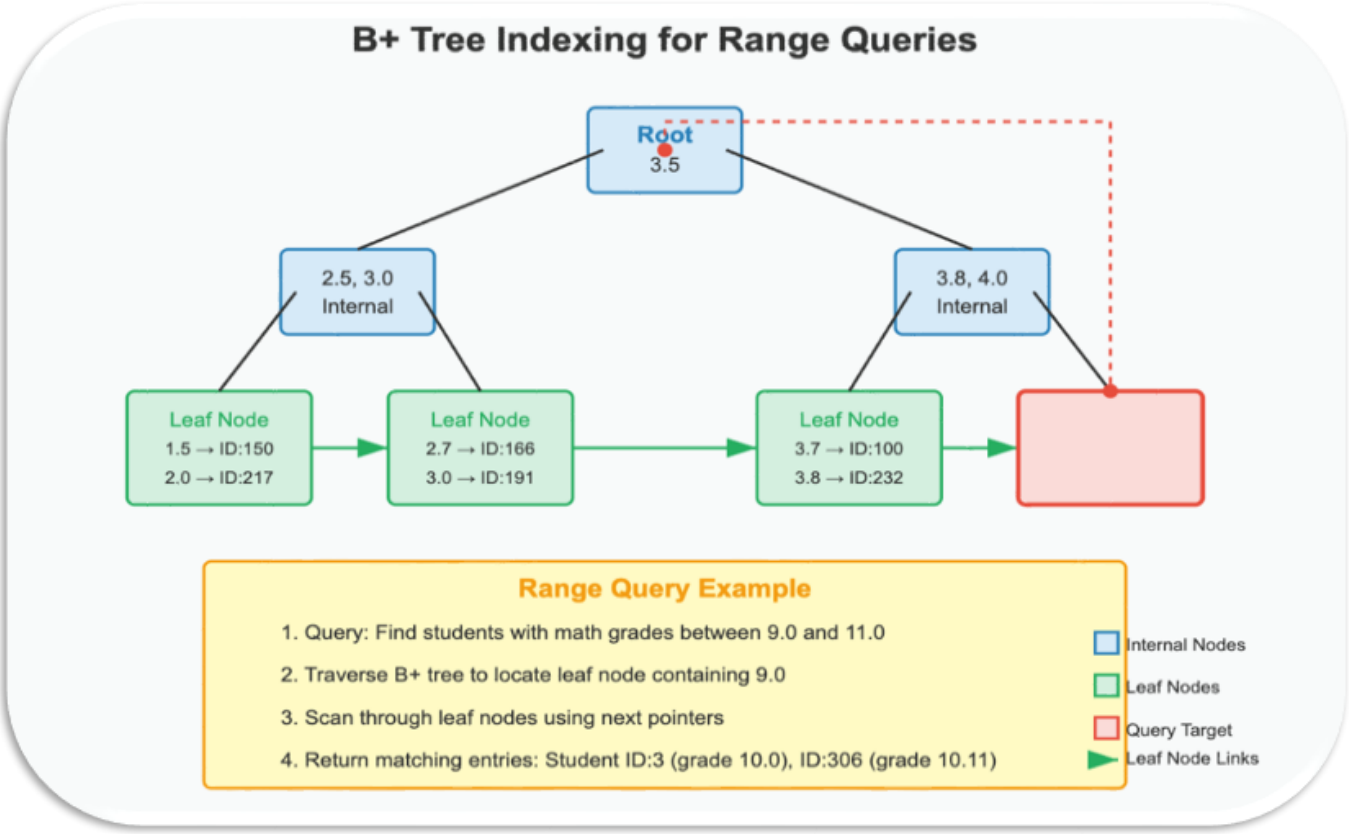
2. The data is divided into k equal chunks
3. Reed-Solomon encoding generates n-k additional parity chunks
4. Each of the n total chunks is packaged with metadata as a fragment
5. Fragments are distributed across different nodes

For our implementation, we use parameters k=3 and n=5, meaning any 3 out of 5 fragments are sufficient to reconstruct a block, providing 40% redundancy while reducing per-node storage requirements by 60%.

### *4.3 B+ Tree Indexing*

To support efficient range queries, we implement a B+ tree index structure that maps attribute values to block and record identifiers. The B+ tree is particularly well-suited for range queries due to its ordered leaf nodes with sibling pointers.

Our implementation includes:
1. A configurable order parameter to optimize node capacity
2. Support for duplicate keys through value lists
3. Efficient range query operations that traverse only relevant leaf nodes




## Table 1. B+ Tree Performance Characteristics

| Operation | Time Complexity | Space Efficiency | Query Capability |
|---|---|---|---|
| Insert | O(log n) | High | N/A |

| | | | |
|---|---|---|---|
| Point Query | O(log n) | N/A | Good |
| Range Query | O(log n + m) | N/A | Very Good |

Where n is the number of keys and m is the number of keys in the result range.

## 5. IMPLEMENTATION DETAILS

### *5.1 System Components*

Our implementation is structured as a modular Python application with the following main components:

- *Blockchain Module: Implements the core blockchain data structure with block creation, validation, and chain management.*
- *Coding Module: Provides encoder and decoder classes that implement the Reed-Solomon coding scheme for fragmenting and reconstructing blocks.*
- *Indexing Module: Implements the B+ tree structure for efficient data indexing and range queries.*
- *Storage Module: Manages distributed storage across multiple nodes, handling fragment placement and retrieval.*

Blockchain Module: Implements the core blockchain data structure with block creation, validation, and chain management.

Coding Module: Provides encoder and decoder classes that implement the Reed-Solomon coding scheme for fragmenting and reconstructing blocks.

Indexing Module: Implements the B+ tree structure for efficient data indexing and range queries.

Storage Module: Manages distributed storage across multiple nodes, handling fragment placement and retrieval.

## 6. IMPLEMENTATION

### *6.1 Design*

The system follows a layered architecture with clear separation of concerns:

- The blockchain layer maintains the chain structure and block validation
- The coding layer handles block fragmentation and reconstruction
- The indexing layer provides efficient query capabilities
- The storage layer manages the distributed node network

This design enables independent scaling and optimization of each component while maintaining the integrity of the overall system. The communication between layers follows a well-defined API, allowing for future extensions and modifications.

### *6.2 Description of the code/script*

The implementation consists of several key Python modules:

- *blockchain/blockchain.py: Implements the Block and Blockchain classes for creating and managing the blockchain.*
- *coding/encoder.py: Implements the Encoder class that fragments blocks using Reed-Solomon codes.*
- *coding/decoder.py: Implements the Decoder class that reconstructs blocks from fragments.*
- *indexing/bplus_tree.py: Implements the BPlusTree and BPlusTreeNode classes for indexing and range queries.*
- *storage/node_server.py: Implements the NodeServer class representing individual storage nodes.*
- *storage/node_manager.py: Implements the NodeManager class for coordinating multiple nodes.*
- *storage/distributed_store.py: Implements the DistributedStore class for storing and retrieving fragments.*
- *main.py: The entry point that demonstrates the entire system workflow.*

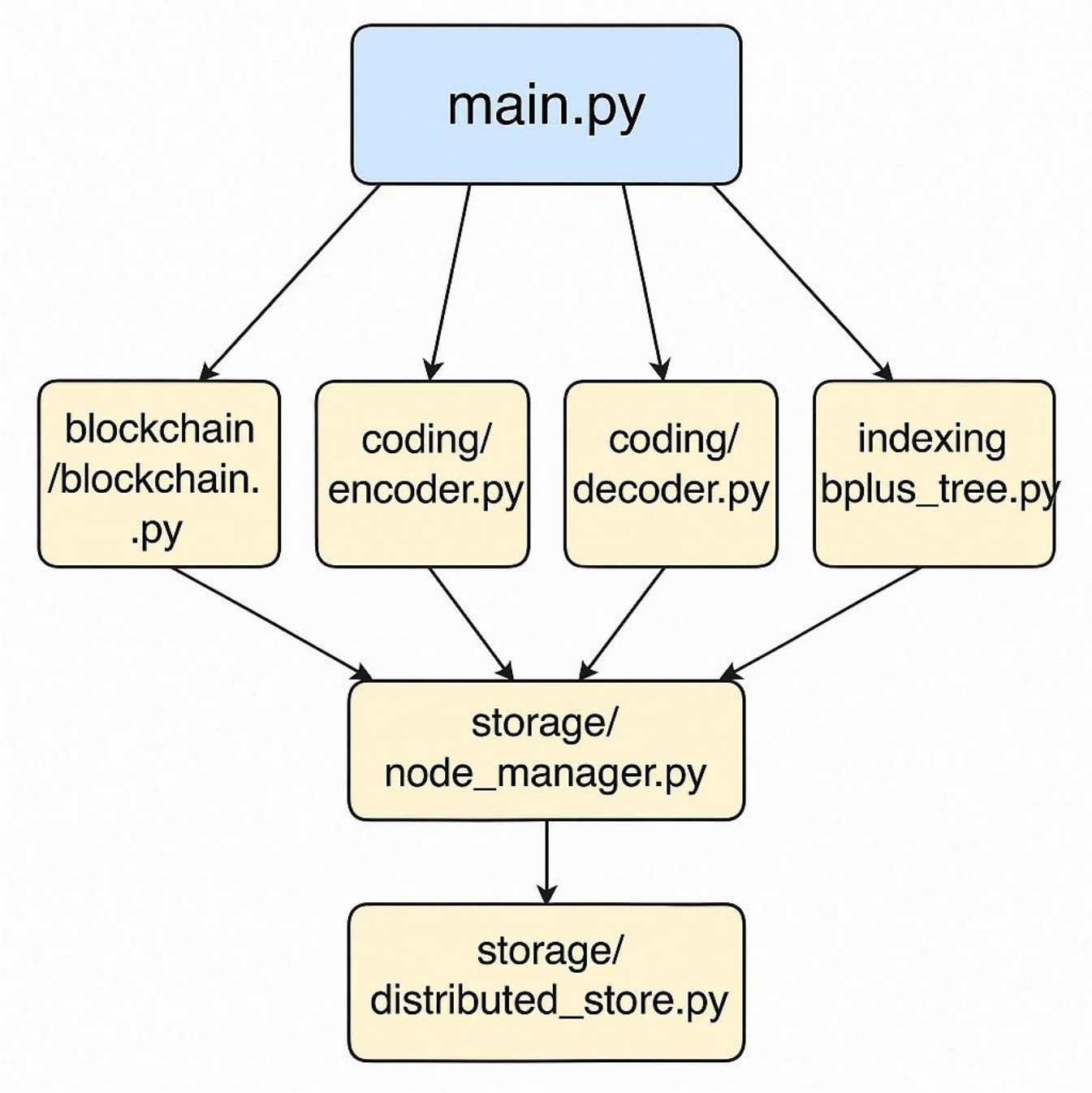



Figure 1. Architecture

The Node class uses Flask to expose REST endpoints for fragment storage and retrieval, while the BPlusTree implementation provides efficient range query capabilities. Error correction coding is implemented using the zfec library, which provides Reed-Solomon encoding/decoding functionality.

### *6.3 Deterministic algorithm for fragment distribution*

The fragment distribution in our coded blockchain system uses a deterministic algorithm to ensure balanced and predictable placement across storage nodes. Each fragment, generated using Reed-Solomon encoding, is assigned a unique ID based on the block hash and fragment index. A consistent hashing function **(e.g., hash(fragment_id) % num_nodes)** is applied to map each fragment to a specific node. This mapping ensures uniform distribution without needing a central coordinator and allows all nodes to independently compute where a fragment should be stored or retrieved from.

The NodeManager maintains a static list of active nodes, and the DistributedStore uses this list during encoding and retrieval to route fragments appropriately. During decoding, the required fragments are requested based on the same deterministic mapping logic. This approach provides high availability and fault tolerance, as blocks can be reconstructed from any subset of k out of k + r fragments, minimizing data loss while maintaining efficient storage utilization.

## 7. EVALUATION

### *7.1 Experimental setup*

We evaluated our system using a dataset of student records (students.csv) containing 306 entries with various attributes including grades, demographic information, and ratings. The evaluation was performed on a simulated network of 6 nodes, with blocks containing 10 records each.

The system was configured with the following parameters:
- Blockchain block size: 10 records per block
- Coding parameters: k=3 (data fragments), redundancy=2 (parity fragments)
- B+ tree order: 10
- Number of nodes: 6

Our evaluation focused on:
1. Storage efficiency compared to traditional blockchain
2. Range query performance on indexed attributes
3. Data recovery capability under node failure scenarios

### *7.2 Experimental results*

Our evaluation yielded the following key results:

- Storage Efficiency: Each node stored approximately 33% of the total blockchain data while maintaining full data recoverability. This means each node stores roughly 60% less than it would in a traditional full-chain system, while still allowing full block reconstruction from any 60% of fragments.
- Range Query Performance: Range queries on indexed attributes demonstrated logarithmic time complexity relative to the number of records. For example, a query on math grades between 9.0 and 11.0 successfully retrieved matching records by traversing only relevant blocks via the B+ tree index, avoiding a full-chain scan.

- Fault Tolerance: The system successfully recovered complete blocks when retrieving any 3 out of 5 fragments, confirming the theoretical guarantees of the Reed-Solomon coding scheme. This demonstrates robustness against up to 40% node failures.
- Scalability: As we increased the dataset size, the storage savings remained consistent at approximately 60%, while query performance scaled logarithmically with respect to the number of records.

These results confirm that our approach successfully addresses the core challenges of storage efficiency and query performance in blockchain systems.

| Feature | Traditional Blockchain | Coded Blockchain with B+ Tree |
|---|---|---|
| Query Execution | Linear scan of all blocks | Indexed block access via B+ Tree |
| Query Time Complexity | O(n) | O(log n) + O(k) (k = matched items) |
| Data Privacy | Plaintext or externally encrypted | Built-in encoding before storage |
| Index Support | None | B+ Tree indexing |
| Range Queries | Inefficient or unsupported | Efficient via leaf node linking |
| Scalability (Data Size) | Poor (slows down as data grows) | Better due to indexing and distribution |
| Storage Architecture | Monolithic | Distributed across simulated nodes |
| Memory Usage | High (raw & repeated data stored) | Lower (encoded, compact, non-redundant) |

## 8. DEMO

Our demonstration showcases the full workflow of the coded blockchain system:

### 1. Starting up flask servers

```
venvdineth@Dineths-MacBook-Pro coded-blockchain-query % python src/main.py
Node server node_1 started on 127.0.0.1:5001
 * Serving Flask app 'node_node_1'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5001
Press CTRL+C to quit
Node server node_2 started on 127.0.0.1:5002
 * Serving Flask app 'node_node_2'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5002
Press CTRL+C to quit
Node server node_3 started on 127.0.0.1:5003
 * Serving Flask app 'node_node_3'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
```

### 1. Creating blockchain blocks with student records

```
=== Loading data from CSV ===

=== Creating Blocks and Adding them to blockchain ===
Creating block 0 with 10 records
Creating block 1 with 10 records
Creating block 2 with 10 records
Creating block 3 with 10 records
Creating block 4 with 10 records
Creating block 5 with 10 records
Creating block 6 with 10 records
Creating block 7 with 10 records
Creating block 8 with 10 records
Creating block 9 with 10 records
Creating block 10 with 10 records
```

1. **Encoding blocks into fragments**
1. **Distributing fragments across nodes**

```
=== Encoding and distributing blocks ===
Encoding block: 0
Distributing 5 fragments across 6 nodes
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/0/0 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/0/1 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/0/2 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/0/3 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/0/4 HTTP/1.1" 200 -
Encoding block: 1
Distributing 5 fragments across 6 nodes
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/1/0 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/1/1 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/1/2 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/1/3 HTTP/1.1" 200 -
127.0.0.1 - - [31/Mar/2025 22:51:51] "PUT /fragment/1/4 HTTP/1.1" 200 -
Encoding block: 2
```

1. **Building a B+ tree index on the math.grade attribute**
1. **Performing a range query (finding students with math grades between 9.0 and 11.0)**
1. **Retrieving and reconstructing blocks containing the query results**

```
=== Building B+ tree index for math.grade ===

=== Executing grade range query using B+ Tree ===
Students with math grades 9.0 - 11.0: 2

=== Found matching students in 2 blocks ===

Retrieving fragments for block: 1
Student IDs to retrieve: ['3']
Attempting to retrieve fragment 0 from node_2...
127.0.0.1 - - [31/Mar/2025 22:51:51] "GET /fragment/1/0 HTTP/1.1" 200 -
Successfully retrieved fragment 0 from node_2
Attempting to retrieve fragment 1 from node_3...
127.0.0.1 - - [31/Mar/2025 22:51:51] "GET /fragment/1/1 HTTP/1.1" 200 -
Successfully retrieved fragment 1 from node_3
Attempting to retrieve fragment 2 from node_4...
127.0.0.1 - - [31/Mar/2025 22:51:51] "GET /fragment/1/2 HTTP/1.1" 200 -
Successfully retrieved fragment 2 from node_4
Block 1 integrity verified
ID: 3, Name: Natasha Yarusso, Math Grade: 10.0

Retrieving fragments for block: 31
Student IDs to retrieve: ['306']
Attempting to retrieve fragment 0 from node_2...
127.0.0.1 - - [31/Mar/2025 22:51:51] "GET /fragment/31/0 HTTP/1.1" 200 -
Successfully retrieved fragment 0 from node_2
Attempting to retrieve fragment 1 from node_3...
127.0.0.1 - - [31/Mar/2025 22:51:51] "GET /fragment/31/1 HTTP/1.1" 200 -
Successfully retrieved fragment 1 from node_3
Attempting to retrieve fragment 2 from node_4...
127.0.0.1 - - [31/Mar/2025 22:51:51] "GET /fragment/31/2 HTTP/1.1" 200 -
Successfully retrieved fragment 2 from node_4
Block 31 integrity verified
ID: 306, Name: Dane Whittemore, Math Grade: 10.11
venvdineth@Dineths-MacBook-Pro coded-blockchain-query %
```

The demo output confirms successful block encoding, distribution, and recovery, as well as efficient range query execution. For example, the system correctly identifies students with exceptionally high math grades (including

those with grades of 10.0 or higher), retrieves their fragments from the distributed nodes, reconstructs the blocks, and displays the student information.

## 9. CHALLENGES

One of the primary challenges we faced was handling errors during the use of the zfec library for Reed-Solomon encoding and decoding. The library, while powerful, occasionally produced low-level issues such as mismatched fragment lengths or type misalignment. Resolving these issues required diving into the internals of the library and gaining a deeper understanding of how fragment sizes and input buffers were managed.

Another significant challenge was the attempted integration of the **Authenticated Multi-Version Skip List (AMVSL)**, a data structure introduced by Linoy, Ray, and Stakhanova in their IEEE paper *"Authenticated Multi-Version Index for Blockchain-based Range Queries on Historical Data."* AMVSL is designed to support efficient and tamper-evident range queries on versioned blockchain data, aligning closely with our system's goals. However, due to the complexity of the structure and the learning curve involved, we were unable to complete its implementation within the scope of the project. Despite this, studying AMVSL enriched our understanding of secure indexing mechanisms for historical blockchain data.

## 10. INDIVIDUAL CONTRIBUTION

Our team members contributed to different aspects of the project:

Brahmpreet Singh:
- Implemented the blockchain core module
- Designed the node communication protocol
- Implemented range query functionality
- Contributed to the system evaluation

Dineth Mudugamuwa Hewage:
- Implemented the error correction coding module (encoder/decoder)
- Developed the distributed storage system
- Developed the B+ tree indexing structure
- Led the integration of system components

Nicholas George Allison:
- Implemented the node server and manager
- Developed the system evaluation framework
- Created the data loading and processing utilities
- Authored the project documentation and report

All team members participated in the design discussions, system testing, and performance evaluation.

## 11. CONCLUSIONS

This project demonstrates that combining error correction coding with efficient indexing structures provides a viable solution to the storage and query challenges in blockchain systems. Our coded blockchain implementation achieves significant storage savings while maintaining data integrity and enabling efficient

range queries on historical data. Our system balances storage savings and query performance through the practical use of Reed-Solomon coding and B+ tree indexing.

Key contributions of our work include:

1. A novel approach to blockchain storage using error correction codes
2. An efficient B+ tree indexing structure for range queries
3. A distributed architecture that balances storage efficiency and fault tolerance

Future work could explore:

1. Dynamic adjustment of coding parameters based on network conditions
2. Support for complex queries beyond simple range operations
3. Integration with consensus mechanisms for indices
4. Optimization for specific application domains like IoT or financial systems
5. Using a P2P network for nodes.

Our approach opens new possibilities for blockchain applications in resource-constrained environments where storing the entire blockchain is impractical but maintaining data integrity and query capabilities remains essential.

## 12. REFERENCES


[1] Dryja, T. 2016. *Utreexo: A dynamic hash-based accumulator optimized for the Bitcoin UTXO set.* Cryptology ePrint Archive, Report 2019/611.

[2] Linoy, S., Ray, S., and Stakhanova, N. 2022. *Authenticated Multi-Version Index for Blockchain-based Range Queries on Historical Data.* IEEE International Conference on Blockchain (Blockchain).

[3] Miller, A., Hicks, M., Katz, J., and Shi, E. 2014. *Authenticated Data Structures, Generically.* In Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14).

[4] Student Data Set: https://github.com/ShapeLab/ZooidsCompositePhysicalizations/blob/master/Zooid_Vis/bin/data/student-dataset.csv

[5] Tahoe-LAFS Project. *zfec: fast erasure coding library.* GitHub repository. Available at: https://github.com/tahoe-lafs/zfec

[6] Van Flymen, D. 2017. *A simple implementation of blockchain in Python.* GitHub repository. Available at: https://github.com/dvf/blockchain/blob/master/blockchain.py

[7] Weatherspoon, H., and Kubiatowicz, J. D. 2002. *Erasure Coding vs. Replication: A Quantitative Comparison.* In Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS '01).

[8] Wöhrer, M., and Zdun, U. 2018. *Smart contracts: Security patterns in the ethereum ecosystem and solidity.* In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE).

[9] Xu, C., Zhang, C., and Xu, J. 2019. *vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases.* In Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19).