Leveraging Binlog for Efficient Blockchain-based Database Synchronization

Kaipeng Wang Binghao Huang Jiawei Hu Yisheng Zhu

Abstract

This research project introduces a novel approach that integrates binary log (binlog) technology from traditional databases into a new blockchain-based Database application, where each node maintains its own independent database. By utilizing binlog to track and record database changes, and synchronizing these changes across nodes via the blockchain, the system can optimize on-chain data storage and communication, improve overall throughput, and maintain data integrity. Additionally, this research explores the performance bottlenecks of traditional JSON formats in large-scale node networks, with a focus on optimizing data serialization methods to address the performance challenges faced by existing blockchain systems in high-concurrency scenarios.

1 Project Overview

This documentation covers two related blockchain projects:

1.1 Blockchain Database using Binlog

Project Repository: https://github.com/Carpe-Wang/BlockChain-Py

1.2 Binlog Performance Testing

Test Repository: https://github.com/Carpe-Wang/Blockchain-Draft

2 Introduction

Blockchain technology, as a decentralized distributed ledger technology, relies on a Peer-to-Peer (P2P) network and the data transmission and synchronization mechanisms between nodes in the network. However, in practical applications, communication among multiple nodes in a network faces many challenges and issues [3]. According to relevant research literature, these problems mainly include network latency, data redundancy, excessive bandwidth consumption, and low node synchronization efficiency. Particularly in large-scale blockchain networks, as the number of nodes increases, these problems are exacerbated, severely affecting the overall performance and scalability of the system.

Among these, the choice of data transmission format is one of the key factors influencing network transmission efficiency [18]. Traditional text formats such as JSON or XML, although having good readability, are highly redundant [4]. In scenarios involving massive data transmissions, they can lead to excessive consumption of bandwidth resources. To address this issue, this study proposes introducing binary log (binlog) technology into blockchain networks [20]. Binlog, as an efficient binary format for recording data, has the advantages of compact data structures and high transmission efficiency.

This research aims to explore the application of binlog in blockchain networks, optimizing data serialization methods to achieve a more efficient inter-node communication mechanism. This improvement not only holds the potential to enhance data transmission efficiency in blockchain networks but also provides new insights and solutions for addressing performance bottlenecks in large-scale blockchain systems.

3 Current technological shortcomings

Using BigChainDB as an example [11] [19], the data being transmitted is primarily in JSON format: A BigchainDB transaction is a JSON string that conforms to a BigchainDB Transactions Specification (Spec).

However, JSON involves the following issues:

Relational Databases [9]:

Data Serialization and Deserialization: JSON data needs to be serialized when transmitted and deserialized when read. This process introduces a certain level of performance overhead.

Redundant Fields: JSON typically includes field names and structural information. When dealing with large amounts of data, this metadata can significantly increase the burden on storage and transmission.

Challenges with High Throughput: Due to the overhead of serialization and deserialization, the JSON format is not efficient for handling high-throughput scenarios. In situations with high concurrency, if the system cannot handle a large number of transaction requests simultaneously, the network transmission speed will be limited. Optimizing concurrent processing capabilities, such as through parallel validation or sharding techniques, can improve system throughput and speed [15].

NoSQL Databases [7]:

Cache Space Utilization: In blockchain technology, NoSQL databases often use caching (e.g., Redis) to speed up data queries. The presence of redundant fields in JSON data can cause each data entry to occupy more cache space than the actual content, leading to inefficient use of cache capacity.

Cache Performance Degradation: When large amounts of JSON data are stored in the cache, the cache hit rate can decrease. Especially when the cache approaches its capacity limit, the redundant JSON data may cause frequent data replacement, thereby affecting the overall performance of the system.

Network Transmission Efficiency: Even when data is stored in the cache, the redundant structure of JSON can consume unnecessary bandwidth during data transmission, increasing latency. This issue becomes particularly apparent when synchronizing large amounts of data between blockchain nodes.

4 Network and System Architecture

4.1 Feature of Blockchain Network

Before diving into a micro perspective (how two nodes communicate), some features of blockchain network should be introduced. The P2P network in Blockchain has following features [14] [5]:

1. High redundancy: Every node in the network maintains a complete copy of all historical transactions and blocks. This redundancy, while essential for decentralization and fault tolerance, leads to significant data duplication across the network. For instance, in Bitcoin network, each node stores over 500GB of blockchain data, multiplied by thousands of nodes worldwide.

2. Intensive communication: Nodes constantly exchange various types of data including new transactions, blocks, and network states through flooding mechanism. Each transaction or block needs to be propagated to all nodes in the network, and this process occurs continuously as new data is generated. Such frequent data exchange results in substantial network traffic, especially during peak times.

3. Global scale: The network operates on a worldwide scale with thousands of participating nodes distributed across different geographical locations and varying network conditions. This global distribution introduces challenges such as network latency, bandwidth limitations, and heterogeneous network quality among different regions, making efficient data transmission particularly crucial for maintaining system performance.

So, it's necessary to improve the performance of node-to-node communication in blockchain networks to reduce bandwidth consumption and enhance system scalability.

4.2 System Overview

Having examined the global features of blockchain networks that highlight the significance of data transmission, we now zoom in to investigate how nodes actually communicate with each other. When two blockchain nodes establish a connection, they engage in a series of protocol-defined interactions to exchange blocks, transactions, and network states.

Data synchronization has long been a critical challenge in distributed database systems. Traditional synchronization solutions typically employ JSON-based data exchange formats, which offer good readability and flexibility but present significant performance bottlenecks. JSON formats require serialization and deserialization operations at both sending and receiving ends, consuming substantial computational resources. Moreover, the textual nature of JSON leads to larger data volumes, increasing network transmission overhead. Additionally, JSON formats often contain redundant information such as field names and data type descriptions, further exacerbating storage and transmission burdens (see Figure 1) [11].

Our system proposes an innovative solution by integrating binary log (binlog) technology with blockchain architecture to create an efficient database synchronization mechanism. Binlog, as the native format for recording database changes, offers compact size and fast parsing capabilities. By encapsulating binlog events within a blockchain structure, we achieve both high synchronization efficiency and data consistency (see Figure 2).

The system employs a layered architecture comprising three primary layers: the data layer for database operations and change capture, the blockchain layer for consensus and synchronization, and the network layer for inter-node com-

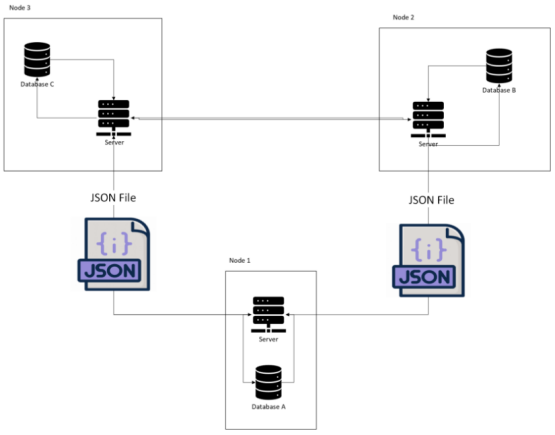

Figure 1: Original Implementation

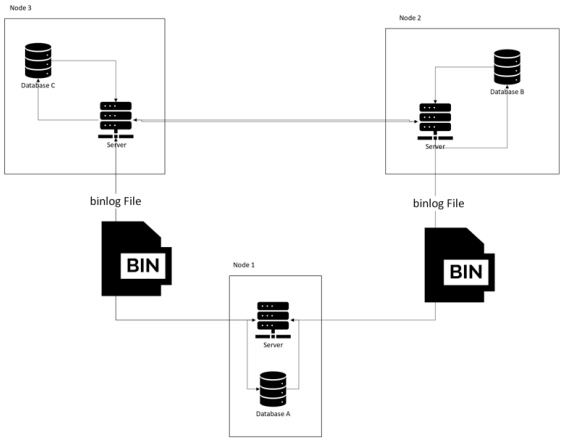

Figure 2: Proposed Implementation

munication. Each layer is designed with clear interfaces and responsibilities, ensuring modularity and maintainability.

- **Data Layer:** Handles database operations and change capture
- **Blockchain Layer:** Manages consensus and synchronization
- **Network Layer:** Facilitates inter-node communication

4.3 Change Data Capture

Our system utilizes MySQL's binlog mechanism [13] for change data capture. By configuring the database's binlog format to `ROW` mode, the system captures detailed and accurate data modifications. Unlike statement-based logging, `ROW` mode records actual data changes rather than SQL statements, ensuring reliable and precise change replication.

The change capture module implements real-time binlog monitoring using MySQL's replication protocol. This approach minimizes overhead while ensuring comprehensive change capture. The system processes various types of database operations, including:

- **DML:** `INSERT`, `UPDATE`, `DELETE` operations
- **DDL:** Table structure modifications
- **Transactions:** `BEGIN`, `COMMIT`, `ROLLBACK` events

For performance optimization, the system implements event batching and filtering mechanisms. Changes are temporarily cached in memory and processed in batches when specific thresholds (time or volume) are reached, reducing system overhead and improving throughput.

4.4 Block Design

The block structure serves as the fundamental data organization unit in our system. Each block is designed to efficiently store and validate database changes while maintaining blockchain properties. The block structure follows a layered design optimized for both storage efficiency and processing speed.

Table 1: Block Structure Design

Component	**Size (Bytes)**	**Purpose**
Previous Hash	32	Chain integrity
Timestamp	8	Temporal ordering
Change Data	Variable	Actual changes

The block generation process involves several key steps:

1. **Data aggregation:** Collecting change events within a time window.
2. **Block sealing:** Computing the block hash.

4.5 Consensus Mechanism

Our system implements the Practical Byzantine Fault Tolerance (PBFT) consensus algorithm to ensure consistency across all nodes. PBFT can tolerate up to f Byzantine nodes in a system with $3f+1$ total nodes, making it suitable for distributed database environments where some nodes might fail or behave maliciously [8] [12].

The consensus process consists of three main phases:

1. **Pre-prepare phase:** The primary node broadcasts changes with a pre-prepare message to all backup nodes.

2. **Prepare phase:** Backup nodes validate and broadcast a prepare message.

3. **Commit phase:** Nodes broadcast commit messages and execute database changes.

4.6 Workflow Overview

Our system implements a five-stage workflow to achieve efficient and reliable data synchronization across the distributed network [17] [10].

The workflow begins with Change Capture when a database modification occurs. During this stage, the system monitors the database's binlog to capture any changes. Each modification is recorded in the binlog, and our binlog listener captures and transforms these changes into a compact binary format suitable for transmission.

The Event Processing stage focuses on optimizing the captured changes for efficient processing. The system temporarily stores events in a memory buffer and combines related events within a specific time window. A compression mechanism is applied to reduce data size while maintaining data integrity. When either the buffer reaches a predetermined size or a time limit is met, the system proceeds to the next stage.

In the Block Creation stage, the system generates a new block from the processed events. The block includes essential metadata such as a timestamp and the previous block's hash. Once the block is assembled, it is digitally signed and ready for the consensus process. The Consensus stage utilizes the PBFT protocol to achieve agreement among nodes. The primary node broadcasts the new block to all participating nodes, and through PBFT's three-phase process, nodes reach consensus on the block's validity and ordering. This ensures all nodes maintain a consistent view of the data.

Finally, the Synchronization stage involves applying the validated changes. Once consensus is reached, the block is added to the blockchain, and its changes are applied to the local database. The block is then distributed to other nodes in the network, ensuring all nodes maintain the same state.

Through these five stages, the system maintains both efficiency in data transmission and consistency across all nodes in the distributed database system.

5 Synchronization Protocol

5.1 Change Capture Process

5.1.1 Database Operation Monitoring

The system uses dedicated event listeners to monitor database operations in real time, capturing all data modifications. Monitoring parameters can be configured to suit different use cases.

The system monitors database operations through specialized event listeners to capture all data modifications:

Table 2: Monitoring Parameters

Parameter	Value
Polling Interval	10 ms
Batch Size	1000
Filter Rules	Configurable

The purpose of the parameters:

- **Polling Interval:** Polling frequency for database changes.
- **Batch Size:** Maximum number of events per polling.
- **Filter Rules:** Filter by event type, table name, etc., to improve efficiency.

The listener continuously captures incremental changes in the database, generating log entries for subsequent processing. Parameters such as polling interval and batch size can be adjusted based on system throughput and response requirements.

5.1.2 Binlog Entry Generation

Each database modification generates a corresponding *Binlog* entry that contains key information required for accurate replication, including:

- **Operation Type**: Records the type of database operation, such as insert (INSERT), update (UPDATE), delete (DELETE), etc.
- **Table and Fields**: Specifies the table and fields involved in the operation, such as which columns were updated or what data was inserted.
- **Old and New Values**:
  - *Old Value*: For update or delete operations, the Binlog records the old value before the operation for verification.
  - *New Value*: Records the new value after the change, particularly for insert and update operations, to reproduce the changes.
- **Transaction Information**: If the database uses transactions, the Binlog includes transaction ID and commit status to ensure atomicity and consistency across nodes.
- **Timestamp**: Captures the exact moment the operation occurred, essential for maintaining the order of log entries during replay.
- **Unique Identifier**: Each Binlog entry has a unique Log ID to track and identify each operation, preventing duplication or missed synchronization.

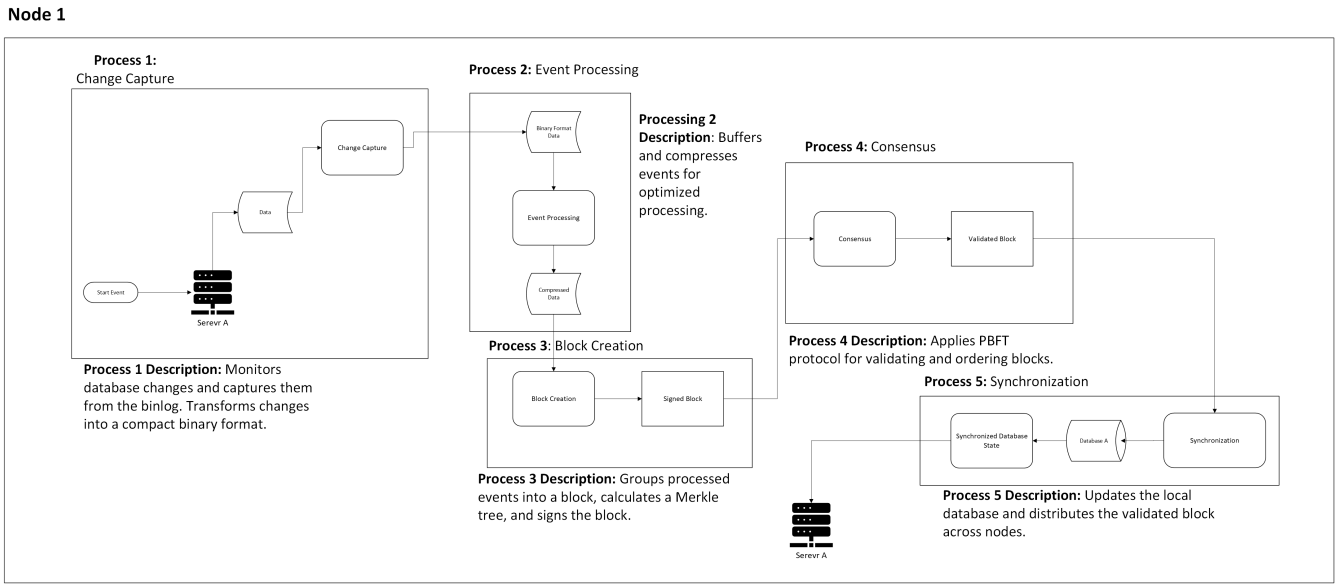

Figure 3: Overall system workflow architecture

- **User/System Source**: Records which user or system initiated the operation (e.g., user ID or service ID) for replay and debugging purposes.
- **Event Sequence Number**: Ensures that events are executed in the correct order, which is critical for maintaining data consistency in high-concurrency scenarios.
- **Data Integrity Check**: Each Binlog entry includes a checksum (e.g., CRC or SHA-256 hash) to ensure data integrity during transmission and storage.

Table 3: Binlog Generation Process

Stage	Time Taken
Operation Capture	0.5 ms
Event Processing	1.0 ms
Format Conversion	0.5 ms
Integrity Check	-

The purpose of each stage:

- **Operation Capture:** Capture database changes and store raw event data.
- **Event Processing:** Process events into standardized Binlog entries.
- **Format Conversion:** Convert Binlog entries into transmission format and verify.
- **Integrity Check:** Validate data to ensure completeness and consistency.

5.1.3 Change Validation

During Binlog entry generation, all changes undergo validation to ensure data integrity and consistency. After generating the Binlog, a series of checks (such as hash verification and data format checks) are performed to prevent data loss or tampering, enhancing system reliability.

5.2 Block Creation and Distribution

5.2.1 Change Aggregation

The system aggregates changes based on configurable parameters to optimize block creation:

Table 4: Aggregation Parameters

Parameter	Value
Time Window	500 ms
Size Limit	1 MB
Event Count	5000

The purpose of each parameter:

- **Time Window:** Maximum block creation interval. Blocks are created every 500 ms.
- **Size Limit:** Maximum block size. Each block is limited to 1 MB.
- **Event Count:** Maximum number of events per block. A block contains up to 5000 events.

5.2.2 Block Assembly

When the aggregation parameters are met, the system assembles the captured events into a block. The assembly process follows the blockchain structure, ensuring data integrity. Each block contains:

- **Block Header**: Includes metadata such as block height, timestamp, and hash of the previous block.
- **Block Body**: Stores a set of aggregated events, specifically a series of Binlog entries.
- **Checksum**: A cryptographic hash ensures that the block data is tamper-proof.

Once assembled, the system performs internal verification to ensure that both the block body and header are complete, so the block can be correctly parsed by other nodes.

5.2.3 Network Distribution

After block assembly, the system broadcasts the block to other nodes via a P2P network. The block propagation mechanism uses shard broadcasting and efficient node discovery algorithms to ensure quick and efficient block distribution. Upon receiving the new block, each node locally verifies the block's validity and updates its blockchain.

5.3 Node Synchronization

5.3.1 Change Validation

Each node verifies the changes upon receiving a new block or incremental data. The validation process ensures data consistency and integrity. Multiple validation metrics and thresholds are set to ensure the reliability of received data.

Table 5: Validation Criteria

Check Type	Threshold
Integrity	100%
Sequence Order	Exact Match
Consistency	-

The action for each check type:

- **Integrity:** Verify checksums to ensure data has not been tampered with.
- **Sequence Order:** Ensure that events are executed in chronological order.
- **Consistency:** Compare block state hash to ensure consistency with on-chain data.

5.3.2 Data Consistency Check

Nodes periodically perform state consistency checks by comparing state hashes or block heights to detect inconsistencies across the network. If inconsistencies are detected, the system triggers correction mechanisms to restore consistency across nodes.

5.3.3 Conflict Resolution

In some cases, conflicts may arise between nodes (e.g., due to concurrent writes or network delays). The system uses deterministic algorithms to resolve these conflicts, ensuring that all nodes reach the same result.

Table 6: Conflict Resolution Priorities

Scenario	Priority
Concurrent Update	1
Missing Event	2
State Mismatch	3

The resolution methods for each scenario are as follows:

- **Concurrent Update:** Timestamp-based sorting to ensure the latest update is applied.
- **Missing Event:** Retry request to re-fetch the missing data.
- **State Mismatch:** Trigger a full data synchronization to resolve inconsistencies.

5.4 Conclusion

This synchronization protocol ensures data consistency and efficient propagation across distributed network nodes through precise Database Change Capture, efficient Binlog Entry Generation, and robust Node Synchronization. The system employs event listening, incremental data processing, block aggregation, and distribution mechanisms to maintain high throughput while ensuring data integrity and consistency in high-performance environments.

6 Experiment and Analysis

6.1 Experiment Design Overview

To evaluate the efficiency of JSON and binlog in data transmission, this study designed a comparative experiment. In this experiment, we used JSON data transmission performance as a benchmark and recorded binlog transmission performance. The detailed experimental procedure is as follows:

First, we implemented a socket-based client-server architecture to establish our testing environment. The client was

configured to serialize data into JSON format before transmission to the server. On the server side, the received JSON data was deserialized and processed accordingly. Throughout this process, we carefully measured and recorded the total time consumed for each operation to establish our baseline metrics.

Following the JSON-based tests, we repeated the same experimental procedure using binary log format. This involved transmitting equivalent data sets using binlog format and processing the received binary data on the server side. As with the JSON tests, we meticulously recorded the total processing time to enable direct comparison between the two approaches.

To ensure statistical reliability and minimize potential errors in our measurements, we implemented a comprehensive testing strategy. Each test scenario was executed in multiple iterations, specifically running 10, 100, and 1000 repetitions. For each set of iterations, we calculated the average time consumption and recorded the standard deviation, providing us with robust metrics to assess the performance stability of both data formats.

As a final component of our analysis, we conducted a detailed comparison of storage efficiency between the two formats. This involved measuring the disk space occupied by varying numbers of records (10, 100, 1000, 10000, 20000, 30000, 40000, 50000, 60000, 70000) in both JSON and binlog formats. And calculating 10 times to calculate the average. Through this comparison, we were able to analyze the space efficiency ratio between the two formats, providing valuable insights into their respective storage requirements.

This experimental design allows us to comprehensively evaluate both the time and space efficiency of JSON versus binlog data transmission methods.

6.2 Implmentation Details

6.2.1 Basic data structure

The following data structure serves as our experimental schema, designed to accurately represent typical database field configurations in a controlled testing environment.

```
type Record struct {
    ID    int    `json:"id"`
    Name  string `json:"name"`
    Value string `json:"value"`
}
```

6.2.2 Test data generator

For comprehensive testing purposes, we developed a random data generation utility. Below is the implementation of our random string generator, which forms the core of our test data generation mechanism.

```
func randomString(length int) string {
    chars := "abcdefghijklmnopqrstuvwxyz" +
             "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
             "0123456789"

    result := make([]byte, length)
    for i := range result {
        result[i] = chars[rand.Intn(len(chars))]
    }
    return string(result)
}

// Generate random record data
func generateRecords(count int) []Record {
    records := make([]Record, count)
    for i := 0; i < count; i++ {

        largeValue := randomString(10 * rand.Intn(10))
        records[i] = Record{
            ID:     i,
            Name:   fmt.Sprintf("name_%d", i),
            Value:  largeValue,
        }
    }
    return records
}
```

- The `randomString` function is designed to generate random strings of a specified length, providing diverse test data for our experiments.
- The `generateRecords` function facilitates the creation of multiple `Record` data structures, each populated with randomized values to simulate real-world data scenarios.
- Within the implementation, `largeValue := randomString(10 * rand.Intn(10))` generates variable-length strings up to 100 characters, effectively simulating diverse data fields typically encountered in database applications.
- This implementation enables us to generate randomized test data that closely resembles real-world database content, ensuring comprehensive performance evaluation.

6.3 Performance Evaluation

In this experimental analysis, we selected serialization time, deserialization time, and file size as key performance metrics. Through controlled variable experiments, we investigated how different factors influence the overall transmission efficiency.

6.3.1 Serialization

```
func jsonSerialize(records []Record) ([]byte, time.Duration) {
    start := time.Now()
    data, _ := json.Marshal(records)
    duration := time.Since(start)
    return data, duration
}

func binlogSerialize(records []Record) ([]byte, time.Duration) {
    var buffer bytes.Buffer
    start := time.Now()
    for _, record := range records {
        binary.Write(&buffer, binary.LittleEndian, int32(record.ID))
```

```
        binary.Write(&buffer, binary.LittleEndian,
            int32(len(record.Name)))
        buffer.WriteString(record.Name)
        binary.Write(&buffer, binary.LittleEndian,
            int32(len(record.Value)))
        buffer.WriteString(record.Value)
    }
    duration := time.Since(start)
    return buffer.Bytes(), duration
}
```

To implement the serialization processes, we developed two primary functions as shown in the code above:

- `jsonSerialize` function: Implements JSON-based data serialization with built-in performance monitoring.
  - Performance measurement: The function utilizes `duration := time.Since(start)` to precisely measure serialization duration, returning both the serialized `data` and its corresponding `duration` for performance analysis.
  - Serialization mechanism: The function employs `json.Marshal(records)` to efficiently convert `records` into JSON byte format. For simplicity in this experimental implementation, error handling is deliberately omitted using the blank identifier (_).
- `binlogSerialize` function: Implements a custom binary log serialization protocol designed for optimal performance.
  - Output and metrics: The function outputs both the serialized binary data (via `buffer.Bytes()`) and the corresponding `duration` metric, enabling direct performance comparison with the JSON implementation.
- Both functions incorporate precise timing mechanisms using `time.Now()` and `time.Since()` to ensure accurate performance measurements for our comparative analysis.

6.3.2 Deserialization

```
// JSON deserialization and timing
func jsonDeserialize(data []byte) (time.Duration, []Record) {
    var records []Record
    start := time.Now()
    _ = json.Unmarshal(data, &records)
    duration := time.Since(start)
    return duration, records
}

// Binlog deserialization and timing
func binlogDeserialize(data []byte) (time.Duration, []Record) {
    buffer := bytes.NewBuffer(data)
    var records []Record
    start := time.Now()
    for buffer.Len() > 0 {
        var id int32
        var nameLen, valueLen int32
        binary.Read(buffer, binary.LittleEndian, &id)
        binary.Read(buffer, binary.LittleEndian, &nameLen)
        name := string(buffer.Next(int(nameLen)))
        binary.Read(buffer, binary.LittleEndian, &valueLen)
        value := string(buffer.Next(int(valueLen)))
        records = append(records, Record{ID: int(id),
            Name: name, Value: value})
    }
    duration := time.Since(start)
    return duration, records
}
```

Complementing our serialization implementation, these deserialization functions perform the inverse operations for both JSON and binary formats. While following the same performance measurement methodology, they transform the serialized byte streams back into their original `Record` structures, enabling us to complete the full data transformation cycle for our performance analysis.

6.3.3 File Size

```
jsonSize = len(jsonData)
binSize = len(binData)
// Record file sizes (convert to KB)
jsonSizes = append(jsonSizes, float64(jsonSize)/1024)
binSizes = append(binlogSizes, float64(binSize)/1024)
```

- The purpose of this code is to calculate the size of JSON data and binary log files, convert them to KB, and record them in the `jsonSizes` and `binSizes` lists for subsequent analysis or processing of file size information.
- `jsonSize = len(jsonData), binSize = len(binData)`: Calculates the byte length of jsonData and binlogData. Assuming jsonData and binlogData are byte arrays or strings, this line returns their size in bytes.
- `jsonSizes = append(jsonSizes, float64(jsonSize)/1024), binSizes = append(binlogSizes, float64(binSize)/1024)`: Divides `jsonSize` and `binSize` by 1024 to convert them to KB and appends them to `jsonSizes` and `binSizes`, respectively. `binlogData` is assumed to contain binary log file data.

6.4 Performance Evaluation and Analysis

6.4.1 Serialization

Our performance evaluation methodology employs a comparative benchmark approach, quantified by the following improvement ratio formula:

$$Speedeup = \left(\frac{Time_{JSON}/Size_{JSON} - Time_{binlog}/Size_{binlog}}{Time_{JSON}/Size_{JSON}} \right) \times 100\%$$

This metric serves as our primary performance indicator, where:

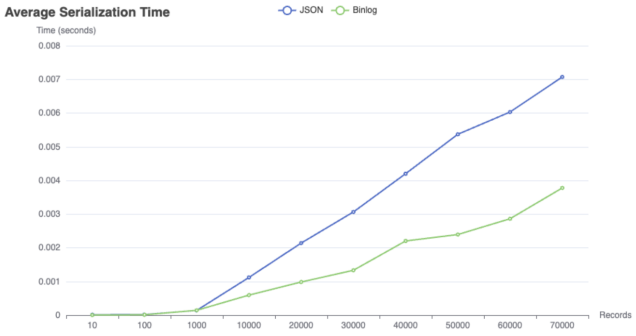

Figure 4: Average Serialization Time

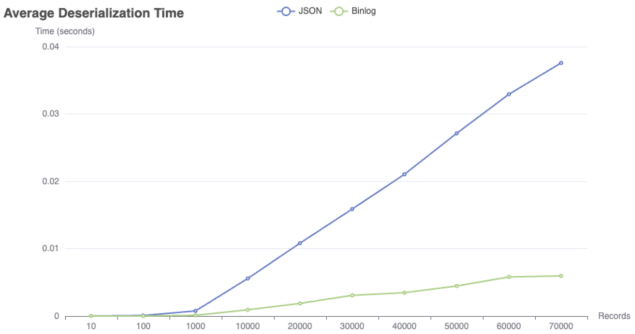

Figure 5: Average Deserialization Time

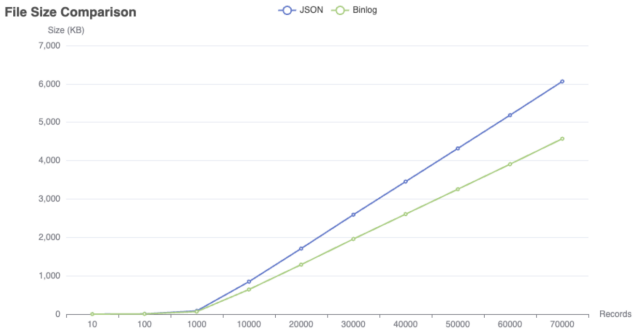

Figure 6: File Size

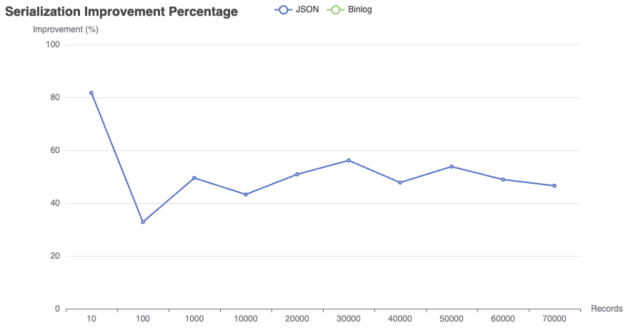

Figure 7: Serialization Improvement

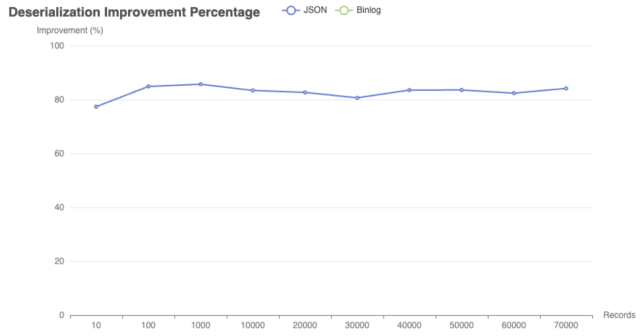

Figure 8: Deserialization Improvement

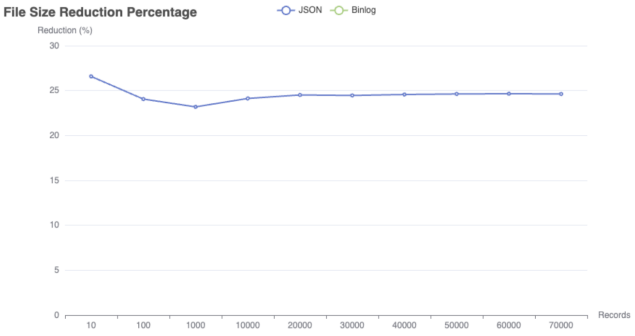

Figure 9: File Size Reduction

- The JSON format serves as the baseline benchmark
- The improvement percentage reflects the relative performance enhancement of binary log format over JSON
- A positive value indicates superior binlog performance
- The ratio of time to size provides a balanced efficiency metric

Based on our experimental results and applying Equation, we observed a significant performance improvement in the serialization process 4 7. The binary log format demonstrated consistently superior efficiency, achieving an average performance enhancement of 50% in processing time compared to the JSON baseline. This substantial improvement suggests that the binary format's compact representation and streamlined encoding process contribute significantly to reduced serialization overhead.

6.4.2 Deserialization

Applying the same evaluation methodology as in the serialization experiments, our analysis of deserialization performance revealed even more substantial improvements. As shown in Figure 5 8, the binary log format demonstrated exceptional efficiency, achieving an improvement of over 80% in processing time compared to the JSON baseline. This notably higher performance gain in deserialization can be attributed to several factors: the streamlined parsing process of binary data, the elimination of syntax validation requirements inherent in JSON processing, and the direct mapping of binary data to memory structures. The more pronounced improvement in deserialization efficiency (80%) compared to serialization (50%)

6.4.3 File Size

From the charts above 6 9, we can observe that compared to JSON, binlog file size is reduced by an average of approximately 25%, enabling faster data synchronization under the same network conditions.

- Network transmission time is mainly affected by file size and network bandwidth. With the same network bandwidth, smaller files occupy the transmission channel for less time, allowing them to reach the destination faster. For larger files, even with the same bandwidth, transmission takes more time to complete. [16]
- Data packet loss and retransmission can occur during network transmission. Larger files are usually split into more data packets, increasing the probability of packet loss, which adds to the number of retransmissions and the overall transmission time. Smaller files contain fewer data packets, and even if packet loss occurs, the retransmission overhead is smaller. [1]
- The network transmission process also involves the processing speed of devices. For example, files may need to be compressed before transmission and decompressed after. Smaller files generally require less processing time for these operations, thus speeding up the overall synchronization process. [6]

7 Real-world Applications

After demonstrating the efficiency of our binlog-based blockchain synchronization approach through theoretical analysis and experimental evaluation, we explore its potential real-world applications. While the system can be adapted for various distributed scenarios requiring strong consistency and low latency, we identify cross-border payment settlement as a particularly promising use case that aligns well with our system's characteristics. The limited number of participating nodes in regional banking networks matches our system's optimal performance characteristics, while the requirement for transaction finality and data consistency demonstrates the advantages of our PBFT-based consensus mechanism combined with binlog synchronization. In this section, we analyze how our system could address current challenges in cross-border payments and discuss potential implementation considerations.

7.1 Cross-border Payment Systems Application Prospect

Cross-border payment settlement represents a potential application scenario for our blockchain-based database synchronization system, particularly in environments with a limited number of participating nodes. Traditional cross-border payment systems typically involve complex intermediary chains and time-consuming reconciliation processes, resulting in high operational costs and settlement delays [2]. Our system shows promise in bringing significant improvements to this domain.

7.1.1 Application Concept

In the cross-border payment scenario, each participating bank will operate as an independent node in the network, with an anticipated involvement of 3-10 banks in a regional payment network. This relatively small node scale is particularly well-suited to our system architecture, showing potential to achieve optimal performance while maintaining strong consistency. Each bank will maintain its own database instance, synchronizing transaction records through the binlog mechanism.

According to our design, the system architecture for bank deployment will comprise three main components. The transaction processing layer handles payment requests and performs business rule validation, ensuring all transactions comply with relevant banking requirements and regulatory stan-

dards. The binlog generation component captures all database changes related to payment transactions, including account balance modifications and transaction status updates. The blockchain layer ensures transaction immutability and consensus among participating banks through the PBFT consensus mechanism.

7.1.2 Theoretical Advantages Analysis

In small-scale banking networks, our system design demonstrates significant advantages. Firstly, the PBFT-based consensus mechanism performs exceptionally well with fewer nodes. Unlike consensus mechanisms such as Proof of Work (PoW) or Proof of Stake (PoS) that may produce temporary forks, PBFT provides immediate finality, which is crucial for financial transactions. In a network with n nodes, PBFT can tolerate up to f = (n-1)/3 Byzantine node failures. In our envisioned typical scale of 4-7 banks, the system can maintain normal operation even with 1-2 node failures. [12].

More importantly, our system prevents blockchain forks through the combination of binlog synchronization mechanism and PBFT consensus. This is primarily reflected in strong consistency guarantees: the PBFT consensus mechanism ensures all normal nodes receive and process binlog data in the same order. During each consensus round, nodes first reach agreement on the binlog sequence before generating blocks. This design ensures all nodes always build new blocks on identical database states.

Unlike traditional blockchain systems, our system does not require waiting for multiple confirmation blocks to ensure transaction finality. Once PBFT consensus is reached, state changes recorded in the binlog immediately become final. This immediacy is particularly important for inter-bank settlement. Additionally, the binary format of binlog not only reduces data transmission volume but also provides natural operation order guarantees. Each database change operation carries strict sequence numbers and timestamps, enabling nodes to precisely reproduce database state changes.

The system periodically generates database state snapshot hashes, allowing all nodes to compare these hash values to ensure database state consistency. If inconsistencies are detected, nodes can request complete binlog replay to restore the correct state without forking. In our experimental evaluation, through injecting network delays and node failures in a simulated environment, we verified that the system maintains data consistency under various anomalous conditions. Even in extreme cases, such as network partitions, the system prioritizes security by suspending service until network recovery rather than risking potential forks.

7.1.3 Future Development Directions

Based on our preliminary research, while the system shows promising application prospects in small-scale banking networks, several directions require further exploration. First is system performance optimization, particularly in cross-border scenarios with unstable network conditions. Second is deep integration with existing banking infrastructure, including handling complex multi-currency settlement scenarios. Additionally, although the current design primarily targets small-scale networks, we are also researching how to enhance scalability while maintaining system advantages, preparing for potential future expansion.

Through this potential application scenario analysis, we believe the system is particularly suitable for scenarios with fewer nodes and high security and consistency requirements. While further research and verification are needed, preliminary analysis suggests that the combination of binlog technology and blockchain may provide a new solution for improving cross-border payment infrastructure.

8 Conclusion

This research has presented a novel approach to blockchain-based database synchronization through the integration of binary log technology. By examining the performance of both traditional JSON-based methods and our proposed binlog solution, we have demonstrated significant improvements in data transmission efficiency and system scalability.

The experimental results reveal substantial performance enhancements across multiple critical metrics. In terms of data processing, the binary log format achieved a 50% reduction in serialization time compared to JSON serialization, while deserialization performance showed an even more remarkable 80% improvement. These gains can be attributed to the streamlined nature of binary data processing and the elimination of complex parsing requirements inherent in JSON handling.

Storage efficiency also showed notable improvement, with the binary log format requiring approximately 25% less space than equivalent JSON representations. This reduction in data volume has direct implications for network transmission efficiency, particularly in large-scale blockchain networks where bandwidth utilization is a critical concern.

The practical implications of these improvements extend beyond mere performance metrics. In high-throughput scenarios where database synchronization is frequent and time-sensitive, the reduced processing overhead and improved data compression contribute to more responsive and scalable systems. Furthermore, the maintenance of data integrity through blockchain verification mechanisms ensures that these performance gains do not come at the cost of reliability or consistency.

Looking forward, this research opens several avenues for future investigation. The integration of more sophisticated compression algorithms, the exploration of adaptive serialization strategies based on network conditions, and the optimization of consensus mechanisms specifically for binary

log data all represent potential areas for further improvement. Additionally, the principles demonstrated in this study could be extended to other distributed database applications where efficient data synchronization is crucial.

In conclusion, our binary log-based approach represents a advancement in blockchain-based database synchronization, offering a more efficient and scalable solution while maintaining the robust security and consistency guarantees that blockchain technology provides. These improvements contribute to the broader goal of making blockchain databases more practical and performant in real-world applications.

References

[1] Runa Barik, Michael Welzl, Peyman Teymoori, Safiqul Islam, and Stein Gjessing. Performance evaluation of in-network packet retransmissions using markov chains. In *2020 International Conference on Computing, Networking and Communications (ICNC)*, pages 10–16, 2020.

[2] Morten L Bech, Umar Faruqui, and Takeshi Shirakami. Payments without borders. *BIS Quarterly Review*, pages 53–65, 2020.

[3] Frederick Ehiagwina, Nurudeen Iromini, Ikeola Suhurat Olatinwo, Kabirat Raheem, and Khadijat Mustapha. A state-of-the-art survey of peer-to-peer networks: Research directions, applications and challenges. 1:19–38, 02 2022.

[4] João Eduardo Ferreira, George Spanoudakis, Yutao Ma, and Liang-Jie Zhang, editors. *Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats*, Cham, 2018. Springer International Publishing.

[5] A. B. Goldstein, N. A. Sokolov, V. S. Elagin, A. V. Onufrienko, and I. A. Belozertsev. Network characteristics of blockchain technology of on board communication. In *2019 Systems of Signals Generating and Processing in the Field of on Board Communications*, pages 1–5, 2019.

[6] Feixue Han, Qing Li, Jianer Zhou, Hong Xu, and Yong Jiang. Aps: Adaptive packet sizing for efficient end-to-end network transmission. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2022.

[7] Jing Han, Haihong E, Guan Le, and Jian Du. Survey on nosql database. In *2011 6th International Conference on Pervasive Computing and Applications*, pages 363–366, 2011.

[8] Wenyu Li, Chenglin Feng, Lei Zhang, Hao Xu, Bin Cao, and Muhammad Ali Imran. A scalable multi-layer pbft consensus for blockchain. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1146–1160, 2021.

[9] Zhen Hua Liu. Json data management in rdbms. In Zongmin Ma and Li Yan, editors, *Emerging Technologies and Applications in Data Processing and Management*, pages 20–44. IGI Global, 2019.

[10] Carlo Mastroianni, Giuseppe Pirrò, and Domenico Talia. Data consistency and peer synchronization in cooperative p2p environments. Technical report, Technical Report, unpublished, 2008.

[11] Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, T. Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto. Bigchaindb: A scalable blockchain database. Whitepaper, BigchainDB, 2016.

[12] Jelena Mišić, Vojislav B. Mišić, Xiaolin Chang, and Haytham Qushtom. Adapting pbft for use with blockchain-enabled iot systems. *IEEE Transactions on Vehicular Technology*, 70(1):33–48, 2021.

[13] MySQL. Mysql 8.4 reference manual - the binary log. https://dev.mysql.com/doc/refman/8.4/en/binary-log.html, 2024. MySQL Version 8.4, Accessed: 2024-03-15.

[14] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, USA, 2016.

[15] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. Blockchain meets database: Design and implementation of a blockchain relational database. In *Proceedings of the VLDB Endowment*, volume 12 of *VLDB '19*, 2019.

[16] Ting Qu, Raj Joshi, Mun Choon Chan, Ben Leong, Deke Guo, and Zhong Liu. Sqr: In-network packet loss recovery from link failures for highly reliable datacenter networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*, pages 1–12, 2019.

[17] Peter Reimann, Holger Schwarz, and Bernhard Mitschang. Design, implementation, and evaluation of a tight integration of database and workflow engines. *Journal of Information and Data Management*, 2(3):353–353, 2011.

[18] Felix Martin Schuhknecht, Ankur Sharma, Jens Dittrich, and Divyakant Agrawal. Chainifydb: How to blockchainify any data management system. *arXiv preprint arXiv:1912.04820*, 2019.

[19] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, 2020.

[20] Ahmad Samer Wazan, Romain Laborde, François Barrère, and Abdelmalek Benzekri. Blockchain-based database to ensure data integrity in cloud computing environments. In *Computer Security*, ESORICS 2018, 2018.