data simulation

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

- Use the above data structure to simulate fields in the database.

```go
func randomString(length int) string {
  chars := "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
  result := make([]byte, length)
  for i := range result {
    result[i] = chars[rand.Intn(len(chars))]
  }
  return string(result)
}

// Generate random record data with larger value field
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 generates a random string of a specified length.
- The `generateRecords` function creates a set of `Record` data structures containing random values.
- `largeValue := randomString(10 * rand.Intn(10))` creates a randomly sized string (up to 100 characters) to simulate a larger data field, making it more suitable for database storage.
- Use this code to randomly generate data for the database.

Testing Serialization

```go
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
}
```

- `jsonSerialize` **function**: Serializes data in JSON format.
  - **Calculates and returns serialization time**: `duration := time.Since(start)` calculates the time taken for serialization and returns the serialized data `data` along with `duration`.
  - **Serializes data**: `data, _ := json.Marshal(records)` uses `json.Marshal` to convert `records` into a JSON byte format. Error handling is ignored here (using `_`).
- `binlogSerialize` **function**: Serializes data using a custom binary log format.
  - **Returns data and serialization time**: Upon completion, returns the binary data in `buffer.Bytes()` and the time taken for serialization `duration`.
- Records the start and end time of each operation to calculate execution duration.

Testing Deserialization

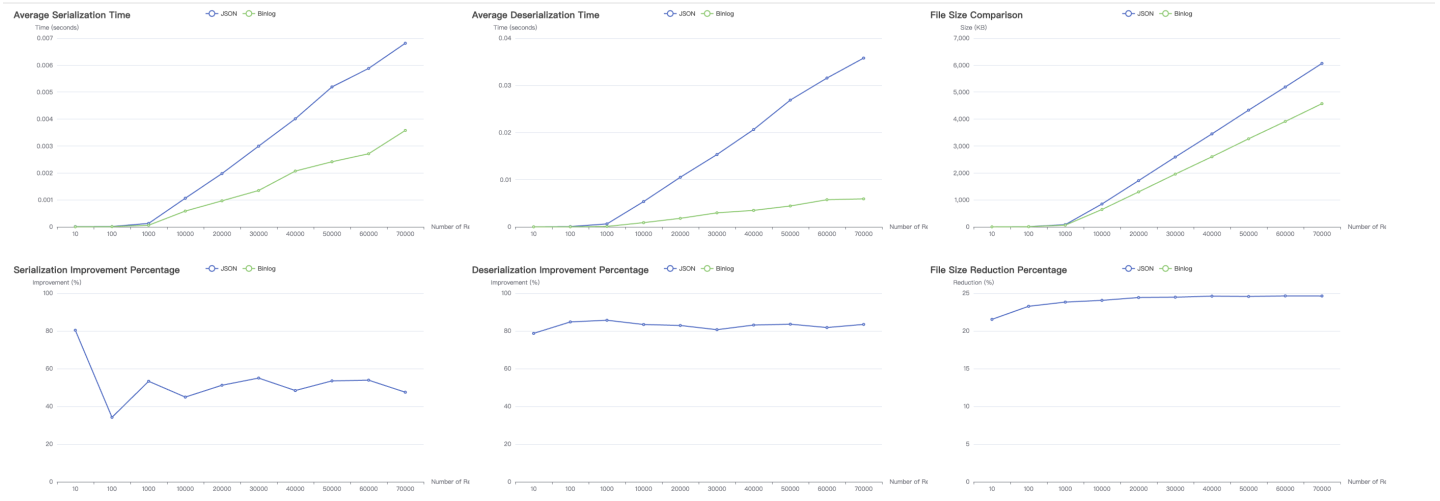
- Same meaning as the serialization method

  ```go
  // 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
  ```

```go
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
}
```

Analyze Data



- According the below formula
-

$$\text{improvement} = \left(\frac{\text{JSON time/size} - \text{Binlog time/size}}{\text{JSON time/size}}\right) \times 100 \qquad (1)$$

- to calculate the improvement
- **Serialization:**
  - We can observe that serialization can improve performance by an average of 50% in terms of time.
- **Deserialization**:
  - We can observe an improvement of over 80% in time.

File Size

- 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. ----source
- 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. ----source
- 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. ----source
- From the charts above, 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.