JavaScript Code

High-performance JavaScript code for a complex data processing task.
This sample uses advanced concepts including closures, asynchronous programming,
array manipulations, functional programming, and ES6+ features to process and
analyze data.
It demonstrates best practices for code optimization, modularity, and maintainability.

```
class DataProcessor {
  constructor(data) {
    this.data = data; // Initial data
    this.processedData = []; // Placeholder for processed data
  }

  // Method for filtering the data based on custom criteria
  filterData(criteria) {
    return this.data.filter(item => {
      return item.value >= criteria.min && item.value <= criteria.max;
    });
  }

  // Method for transforming the data (e.g., adding a processed flag)
  transformData(transformationFunction) {
    this.processedData = this.data.map(item => transformationFunction(item));
  }

  // Method for reducing the data to a summary
  reduceData(reductionFunction, initialValue) {
    return this.data.reduce(reductionFunction, initialValue);
  }

  // Method for batching the data to optimize processing (useful for large datasets)
  batchProcess(batchSize, processFunction) {
    const batches = [];
    for (let i = 0; i < this.data.length; i += batchSize) {
      batches.push(this.data.slice(i, i + batchSize));
    }
```

```
    // Process each batch asynchronously
    return Promise.all(
      batches.map(batch =>
        new Promise(resolve => {
          setTimeout(() => {
            const processedBatch = batch.map(processFunction);
            resolve(processedBatch);
          }, 0);
        })
      )
    ).then(results => {
      return results.flat(); // Flatten the array after processing all batches
    });
  }

  // Demonstrating usage of async/await and closures for optimization
  async processDataAsync(criteria, transformationFunction, reductionFunction,
initialValue) {
    try {
      const filteredData = this.filterData(criteria);

      // Using batch processing for large datasets
      const transformedData = await this.batchProcess(100, transformationFunction);
      this.processedData = transformedData;

      // Reduce the transformed data to a summary
      const summary = this.reduceData(reductionFunction, initialValue);

      return { transformedData, summary };
    } catch (error) {
      console.error('Error processing data:', error);
      throw error;
    }
  }
}

// Sample data
const sampleData = [
  { id: 1, value: 100, name: 'Item A' },
```

```
  { id: 2, value: 150, name: 'Item B' },
  { id: 3, value: 200, name: 'Item C' },
  { id: 4, value: 50, name: 'Item D' },
  { id: 5, value: 120, name: 'Item E' },
];

// Criteria for filtering
const filterCriteria = { min: 100, max: 200 };

// Transformation function (adds a 'processed' flag)
const addProcessedFlag = item => {
  return { ...item, processed: true };
};

// Reduction function (calculates total value of items)
const calculateTotalValue = (accumulator, item) => {
  return accumulator + item.value;
};

// Usage of DataProcessor class
const processor = new DataProcessor(sampleData);

// Using async function to process data
processor
  .processDataAsync(filterCriteria, addProcessedFlag, calculateTotalValue, 0)
  .then(({ transformedData, summary }) => {
    console.log('Transformed Data:', transformedData);
    console.log('Summary (Total Value):', summary);
  })
  .catch(error => {
    console.error('Error during data processing:', error);
  });

// Function to demonstrate closure usage for data processing
const makeProcessor = multiplier => {
  return function(item) {
    return { ...item, value: item.value * multiplier };
  };
};
```

```
// Applying closure-based transformation
const multiplierProcessor = makeProcessor(2);
const doubledData = sampleData.map(multiplierProcessor);
console.log('Doubled Data:', doubledData);

// Using ES6 Set to handle unique values
const uniqueData = new Set(sampleData.map(item => item.name));
console.log('Unique Data (Names):', [...uniqueData]);

// Efficient sorting using a custom comparator
const sortedData = [...sampleData].sort((a, b) => a.value - b.value);
console.log('Sorted Data by Value:', sortedData);

// Example of immutability in JavaScript
const immutableData = Object.freeze(sampleData);
try {
  immutableData[0].value = 500; // This will throw an error since the object is frozen
} catch (error) {
  console.error('Error modifying immutable data:', error);
}
```

Explanation of Key Concepts:

1. Closures: Used to create functions that remember the scope they were created in, e.g., the makeProcessor function.
2. Asynchronous Programming: Leveraging async/await, Promise.all, and setTimeout to process large datasets efficiently.
3. Functional Programming: Techniques like map(), filter(), reduce(), and using pure functions for transformations.
4. Immutability: Using Object.freeze() to prevent data mutations, making the code safer and reducing bugs.
5. ES6 Features: Using arrow functions, destructuring, and template literals for cleaner, more efficient code.
6. Batch Processing: Efficiently handling large datasets by breaking them into smaller batches, thus improving performance and reducing memory consumption.