

Node.js

Streams in Node.js

Part 1: Introduction & Memory Efficiency

March 22, 2025



Source Code

1. What are Node.js Streams?

Streams in Node.js are abstract interfaces that implement the EventEmitter pattern for handling flowing data. Unlike traditional data processing methods, streams operate on data sequentially in chunks instead of loading entire datasets into memory.

This approach offers three key advantages:

- **Memory Efficiency:** Processing data in small chunks reduces memory consumption, making streams ideal for large files.
- **Time Efficiency:** Operations begin as soon as the first chunks arrive, without waiting for complete data transmission.
- **Composability:** Streams can be connected through piping, enabling complex data processing pipelines.

This architecture naturally aligns with how data flows in many systems, from file I/O to network communications.

1.1. Memory Efficiency: A Practical Example

We'll examine a comprehensive example demonstrating the memory efficiency of streams when processing large files. The example is split into logical parts for better understanding.


1.1.1. Part 1: Setup and Utility Functions

Let's start by setting up the environment and defining utility functions for our comparison:



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```

1  const fs = require('fs');
2  const { promisify } = require('util');
3  const readFile = promisify(fs.readFile);
4
5  // Simple results storage
6  const results = { stream: null, noStream: null };
7
8  // Function to measure memory usage in MB (using RSS for more accurate
   measurement)
9  function getMemoryUsage() {
10   return Math.round(process.memoryUsage().rss / 1024 / 1024 * 100) / 100;
11 }

```

This first part sets up our environment by:

- Importing necessary Node.js modules
- Creating a storage object to hold our comparison results
- Defining a utility function to measure memory usage in megabytes

1.1.2. Part 2: Processing Without Streams

Next, we implement the traditional approach that loads the entire file into memory:

```


1  // Processing WITHOUT streams - loads entire file into memory

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
2 async function processWithoutStreams(filePath) {
3   const initialMemory = getMemoryUsage();
4   console.log('Memory before loading file: ${initialMemory} MB');
5
6   // Read the entire file into memory
7   const data = await readFile(filePath);
8
9   const afterLoadMemory = getMemoryUsage();
10  console.log('Memory after loading file: ${afterLoadMemory} MB');
11  console.log('File size: ${(data.length / 1024 / 1024).toFixed(2)} MB');
12
13  // Process the data (counting lines)
14  const lines = data.toString().split('\n');
15
16  const finalMemory = getMemoryUsage();
17  console.log('Lines processed: ${lines.length}');
18  console.log('Final memory usage: ${finalMemory} MB');
19
20  results.noStream = {
21    initial: initialMemory,
22    afterLoad: afterLoadMemory,
23    max: Math.max(initialMemory, afterLoadMemory, finalMemory),
24    final: finalMemory
25  };
26 }
```

This function demonstrates the traditional approach with these key steps:

- It measures initial memory usage before any operations



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast



www.asanchezyali.com

- It loads the entire file into memory at once using **readFile**
- It measures memory usage after loading the file
- It performs a simple operation (counting lines) on the entire file
- It records final memory usage and stores statistics for comparison

The main disadvantage of this approach is that the entire file must be loaded into memory before processing can begin, which can be problematic for large files.

1.1.3. Part 3: Processing With Streams

Now we implement the stream-based approach that processes data in chunks:

```

1 // Processing WITH streams - processes the file in chunks
2 function processWithStreams(filePath) {
3   return new Promise((resolve) => {
4     const initialMemory = getMemoryUsage();
5     console.log('Memory before starting stream: ${initialMemory} MB');
6
7     let linesCount = 0;
8     let maxMemory = initialMemory;
9     let incompleteLine = '';
10    let chunkCount = 0;
11
12    const readStream = fs.createReadStream(filePath, {
13      highWaterMark: 16 * 1024 // 16KB chunks

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
14     });
15
16     readStream.on('data', (chunk) => {
17         chunkCount++;
18
19         // Process the chunk
20         const data = incompleteLine + chunk.toString();
21         const lines = data.split('\n');
22         incompleteLine = lines.pop();
23         linesCount += lines.length;
24
25         // Track memory usage
26         const currentMemory = getMemoryUsage();
27         maxMemory = Math.max(maxMemory, currentMemory);
28     });
29
30     readStream.on('end', () => {
31         if (incompleteLine) linesCount++;
32         const finalMemory = getMemoryUsage();
33
34         console.log('Lines processed: ${linesCount}');
35         console.log('Total chunks: ${chunkCount}');
36         console.log('Maximum memory used: ${maxMemory} MB');
37         console.log('Final memory usage: ${finalMemory} MB');
38
39         results.stream = {
40             initial: initialMemory,
41             max: maxMemory,
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast



www.asanchezyali.com

```

42         final: finalMemory
43     };
44     resolve();
45 });
46
47     readStream.on('error', (err) => {
48         console.error('Error reading file:', err);
49         resolve();
50     });
51 });
52 }

```

The stream-based approach works differently:

- It creates a read stream with a small buffer size (16KB chunks)
- It processes data incrementally as it arrives in chunks
- It handles line splitting across chunk boundaries with the **incompleteLine** variable
- It tracks maximum and final memory usage throughout processing
- It returns a promise that resolves when processing completes


Key aspects of this implementation:

- The **highWaterMark** option controls the buffer size
- The stream emits «data» events for each chunk



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- We handle incomplete lines that might span across chunks
- Memory usage is tracked continuously during processing

1.1.4. Part 4: Comparison Function and Results

Finally, we implement a function to run both approaches and compare their performance:

```


1 // Main comparison function
2 async function compareMemoryUsage(filePath) {
3   // Get file size for display
4   try {
5     const stats = fs.statSync(filePath);
6     const fileSizeMB = (stats.size / (1024 * 1024)).toFixed(2);
7     console.log('Processing file: ${filePath} (${fileSizeMB} MB)');
8   } catch (err) {
9     console.log('Processing file: ${filePath}');
10  }
11
12  // 1. Run with streams
13  console.log('WITH STREAMS:');
14  await processWithStreams(filePath);
15
16  // Wait for garbage collection
17  console.log('\nWaiting for garbage collection...');
18  await new Promise(resolve => setTimeout(resolve, 2000));
19

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com


NODE.JS | STREAMS

```
20 // 2. Run without streams
21 console.log('WITHOUT STREAMS:');
22 await processWithoutStreams(filePath);
23
24 // Print comparison
25 console.log('\n===== MEMORY USAGE COMPARISON =====');
26 console.log('| Method          | Maximum Memory | Final Memory |');
27 console.log('|-----|-----|-----|');
28 console.log('| WITHOUT STREAM |
    ${results.noStream.max.toFixed(2).padStart(7)} MB    |
    ${results.noStream.final.toFixed(2).padStart(7)} MB    |');
29 console.log('| WITH STREAM    |
    ${results.stream.max.toFixed(2).padStart(7)} MB    |
    ${results.stream.final.toFixed(2).padStart(7)} MB    |');
30 console.log('=====');
31
32 // Memory increase from initial to after loading the file (non-stream
    method)
33 const memoryIncrease = results.noStream.afterLoad -
    results.noStream.initial;
34 console.log('Loading the entire file increased memory by:
    ${memoryIncrease.toFixed(2)} MB');
35
36 // Improvement percentage
37 const improvement = ((results.noStream.max - results.stream.max) /
    results.noStream.max * 100).toFixed(2);
38 console.log('Streams used ${improvement}% less maximum memory!');
39
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```

40 // Tutorial explanation
41 console.log('CONCLUSION:');
42 console.log(' - Streams process data in small chunks (one at a time)');
43 console.log(' - Without streams, the entire file is loaded into
    memory');
44 console.log(' - For large files, streams significantly reduce memory
    usage');
45 }
46
47 // Run the comparison
48 const filePath = process.argv[2] || 'file.txt';
49 compareMemoryUsage(filePath);

```

The comparison function provides a comprehensive analysis:

- It runs both approaches sequentially
- It displays file information before processing
- It adds a delay between runs to allow for garbage collection
- It presents the results in a well-formatted comparison table
- It calculates memory efficiency improvements as a percentage
- It explains the conclusion of the comparison


1.1.5. Comparison Results

When run with a 408MB file, the results are compelling:



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


1 Processing file: file.txt (408.02 MB)
2
3 WITH STREAMS:
4 Memory before starting stream: 36.64 MB
5 Lines processed: 1425071
6 Total chunks: 26114
7 Maximum memory used: 58.98 MB
8 Final memory usage: 56.61 MB
9
10 Waiting for garbage collection...
11
12 WITHOUT STREAMS:
13 Memory before loading file: 57.02 MB
14 Memory after loading file: 459.81 MB
15 File size: 408.02 MB
16 Lines processed: 1425071
17 Final memory usage: 843.95 MB
18
19 ===== MEMORY USAGE COMPARISON =====
20 | Method          | Maximum Memory | Final Memory |
21 |-----|-----|-----|
22 | WITHOUT STREAM | 843.95 MB      | 843.95 MB      |
23 | WITH STREAM    | 58.98 MB       | 56.61 MB       |
24 =====
25
26 Loading the entire file increased memory by: 402.79 MB

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
27 Streams used 93.01% less maximum memory!
28
29 CONCLUSION:
30   - Streams process data in small chunks (one at a time)
31   - Without streams, the entire file is loaded into memory
32   - For large files, streams significantly reduce memory usage
```

The results clearly demonstrate the memory efficiency of streams:

- The non-stream approach used 843.95 MB of memory
- The stream-based approach used only 58.98 MB of memory
- This represents a 93.01% reduction in memory usage
- Both approaches processed the same number of lines (1,425,071)

This example provides compelling evidence for using streams when processing large files in Node.js applications.

2. Conclusions

2.1. Memory Efficiency is Critical for Scalable Applications

Node.js streams provide a powerful solution to memory management challenges in modern applications. As demonstrated in our comparative analysis, processing large datasets with streams can reduce memory consumption by over 90%. This efficiency becomes increasingly important as applications scale and data volumes grow exponentially. By implementing streams in memory-intensive



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast



www.asanchezyali.com

operations, developers can build more resilient systems that maintain performance even under heavy loads.

2.2. Chunked Processing Enables Real-Time Capabilities

The ability to process data in small, manageable chunks creates opportunities for real-time data handling that would be impossible with traditional methods. Rather than waiting for complete datasets to load, stream-based applications can begin processing immediately as data becomes available. This characteristic makes streams particularly valuable for applications requiring low-latency responses, such as real-time analytics, live data visualization, and responsive user interfaces.

2.3. Streams Represent a Shift in Data Processing Paradigms

Adopting streams requires a shift from batch-oriented thinking to event-driven, asynchronous processing models. This paradigm shift aligns perfectly with Node.js's non-blocking I/O philosophy and represents a more natural way to handle data as it flows through a system. By embracing this approach, developers can create more efficient data pipelines that mirror how information naturally moves through modern distributed systems.


3. References

- Node.js. (2023). *Stream* | *Node.js v18.x Documentation*. [Link](#)
- NodeSource. (2022). *Understanding Streams in Node.js*. [Link](#)



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com


NODE.JS | STREAMS

- Alapont, R. (2023). *Streamlining Your Code: Best Practices for Node.js Streams*. [Link](#)
- Alapont, R. (2023). *Error Handling in Node.js Streams: Best Practices*. [Link](#)
- Clarion Technologies. (2022). *Node.js for Real-Time Data Streaming*. [Link](#)
- Translated, Edited and written in collaboration with AI.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

Ready to **Transform** Your Data Flow?

You've discovered the power of Node.js Streams.

Stay tuned for Part 2...

Where we'll unlock the full potential
of your data pipelines!