Porting JSONSki to JavaScript Environments

Gandharva Deshpande

Department of Computer Science, Riverside

Project Advisor: Zhijia Zhao

Table of Contents

Chapter 1

Introduction

JSONSki presents a bit-parallel (SIMD) solution for implementing fast-forward optimizations with high-efficiency to stream over JSON Data originally written in C++. The following are the key differentiators for the JSONSki over other tools that parse JSON data. [1]

- Data Streaming over Querying + Parsing (Tree Parsing)
- SIMD over SISD
- Fast Forward Optimizations over Complete Parse tree

The project aims to port the C++ code to Javascript environments to enhance accessibility.

1.1 Description

JSONSki is a streaming JSONPath processor that was originally developed in C++ and uses SIMD and bitwise operations.[1] In this project, JSONSki was ported to the Javascript running environment such that it can be accessed programmatically via a Node Project, an NPM library and used as an extension tool in VS Code. These three tools make JSONSki more accessible to developers and end users while keeping its performance intact.

1.2 Motivation

JSON is frequently used to transfer data to and from a web server. Data from a web server is always sent to us as a string. The data is parsed using JSON.parse(), and a JavaScript object is created as a result. In a similar vein, data sent to a web server must be a string. With JSON.stringify, we turn a JavaScript object into a string (). Streaming technique eliminates development of data structures when parsing by merging parsing and the query evaluation into a single pass, hence avoiding this memory overhead. JSONSki uses the above technique to efficiently stream and query data using SIMD and fast forward techniques. However, JSONSki written in C++ is infeasible for use to web application developers that predominantly use Javascript (The Language of the Web). Thus, the motivation for

development of a Node.js binding for JSONSki was established. Further, to make it readily available for the developer community, by publishing an open source npm library and a VS Code extension was also essential to enhance ease of use for the tool.

1.3 Proposed Solution

While there are a lot of ways of converting C++ code into a Node.js application, creating a Node.js Add-on offers a ton of flexibility, code reusability and certain performance benefits. A node.js Add-on - that provides interface between C++ and Javascript provides options for add-ons including NAN, N - API (For C), Node Addon-Api (For C++).

Since Node addon API is a part of Node.js, it is documented in-depth on their official documentation. NAN requires rebuilding the module for each NODE_MODULE_VERSION (major version of Node.js) Modules using N-API/Node-Addon-API are forward-compatible. Further, Competitors like SIMDJSON implemented their bindings using Node-addon-API. Hence, Node-addon-API was chosen to create a Node.js binding for JSONSki

1.4 Scope of the Project

The scope of this project comprises of development of the following modules

- NPM library - JSONSKi
- VS Code Extension- jsonski
- CircleCI support for MacOS, Linux - JSONSki C++
- CircleCI support for MacOS - JSONSki_Nodejs
- Benchmarking Repository Node.js (JSON parsing tools in Javascript)
- Benchmarking Repository C++ (JSON parsing tools in C++)
- Github Badges and Documentation for JSONSki C++
- Github Badges and Documentation for JSONSki Node.js

1.5 Literature Review

Zhao and Liang's JSONSKi methodically locates the fast-forward chances in semi-structured data streaming and provides a bit-parallel mechanism for successfully carrying out fast-forward optimizations. Finally, this work verifies the higher performance of the proposed method by contrasting it with a few state-of-the-art JSON data processing tools. Streaming avoids the creation of data structures by combining parsing and query evaluation into a single pass. The trick is to track matches at different record levels using a query stack and syntactic structures using a syntax stack. Developers of ports using other technologies can benefit from JSONSki's performance by adopting fast forward techniques to maintain performance.[1]

Chapter 2

System Analysis

2.1 Hardware and Software Requirements

The hardware requirements of the NPM library and VSCode extension are predominantly compliant with JSONSki C++, however the additional software requirements are due to the presence of Node-addon-API that use node-gyp module written in Python3.

2.1.1 Software requirements:

- Node.JS (v14)
- Python (v3.9)
- C++ : g++ (v7.4.0 and above)

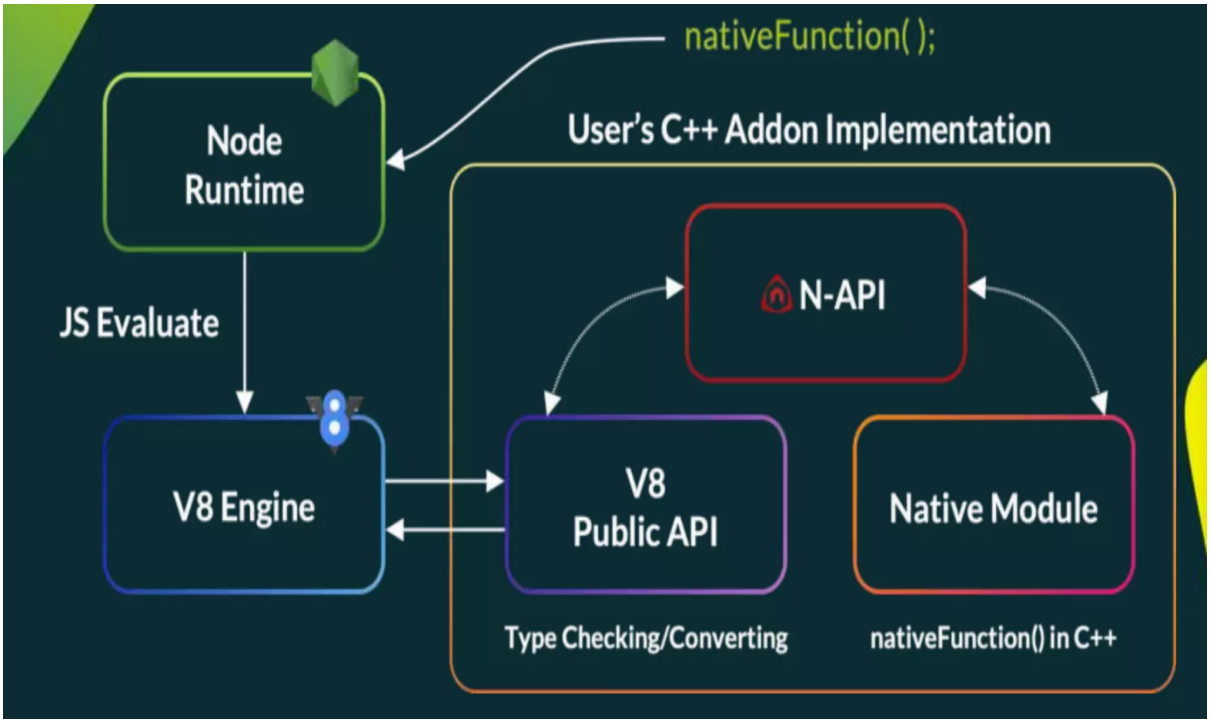
2.1.2 Hardware requirements:

- CPUs: 64-bit ALU instructions, 256-bit SIMD instruction set, and the carry-less multiplication instruction (pclmulqdq)
- Operating System: Linux, MacOs (Intel Architecture)

Chapter 3

Implementation [3-8]

3.1 Node Add on API



Credits: https://www.slideshare.net/nasottola/next-generation-napi

3.1.1 Node-Addon-API [3- 8]

An NPM module called Node-addon-api offers the napi.h header file. This header file, which is officially maintained alongside the N-API and contains C++ wrapper classes for the N-API.

Install Node-addon-api

```
$ npm install -S node-addon-api
```

3.1.2 Binding.gyp

To inform node-gyp about the setup of our Native Module, we use a Python file called binding.gyp that contains a JSON-like data structure. It includes meta-data like as the module name, the list of files that must be assembled, and more.

```
{
  "targets": [{
    "target_name": "JSONSki",
    "cflags!": [ "-fno-exceptions" ],
    "cflags_cc!": [ "-fno-exceptions" ],
    "sources": [
      "src/bindings.cpp",
      "main.cpp",
      "src/RecordLoader.cpp",
      "src/QueryProcessor.cpp",
      "src/JSONPathParser.cpp"
    ],
   "cflags": [
     # Here all your CFLAGS
    ],
    "cflags_cc": [
   # Here all your C ++ FLAGS
    ],
      "xcode_settings": {
```

```
      "OTHER_CFLAGS": ["-O3", "-std=c++11", "-mpclmul", "-lpthread", "-march=native"],
      "GCC_ENABLE_CPP_EXCEPTIONS": "YES"
    },
      'include_dirs': [
          "<!@(node -p \"require('node-addon-api').include\")"
      ],
      'libraries': [],
      'dependencies': [
          "<!(node -p \"require('node-addon-api').gyp\")"
      ],
      'defines': [ 'NAPI_DISABLE_CPP_EXCEPTIONS' ]
   }]
}
```

3.1.3 Details of binding.gyp

The target name, sources, and include directories properties in binding.gyp are crucial to pay attention to. Here, warnings and exceptions from the compiler are disregarded using other properties. A number of Native Addons or DLL files that must be generated are included in the targets list. We are currently only concerned with one DLL file. The name of this DLL file, jsonski, is specified by the target name key.

The sources list contains the list of "src/bindings.cpp," "main.cpp," "src/RecordLoader.cpp," "src/QueryProcessor.cpp," and "src/JSONPathParser.cpp" files that must be built to create the DLL file. The directories that the sources should search for header files in case the compiler is unable to discover them are listed in the include dirs list. To incorporate the SIMD flags, we used the OTHER CFLAGS parameter.

We must specify the location of the napi.h file inside include dirs because we will shortly use it. The `<!@(node -\"require('node-addon-api'). include\")` command is executed by node-gyp to identify the correct directory path of node-addon-api module.

3.1.4 Main.cpp

```
#include <napi.h> // node - api module includes the napi header file so that we can access all the helper macros (#define ls = x.y.z), classes and functions.

#include "src/bindings.h"

Napi::Object InitAll(Napi::Env env, Napi::Object exports) {

  return JsonSki::Init(env, exports);

}

NODE_API_MODULE(JSONSki, InitAll)
```

N-API passes two parameters to InitAll, which takes them.

Env is the first parameter that most N-API functions require, and exports is the object that is used to set the exported functions and classes using N-API. NODE GYP MODULE NAME, INIT, NODE API MODULE

Once the library has been put into active memory, this code instructs Node where to proceed. It defines the entry-point for the Node addon. The first argument must match the "target" in our binding.gyp. As long as the module was created using node-gyp, using NODE GYP MODULE NAME guarantees that the argument will be accurate (which is the usual way of building modules). The function to call is indicated by the second argument. There can be no namespacing in the function.

3.1.5 Bindings.cpp

```
#include "bindings.h"

#include "../src/RecordLoader.h"

#include "../src/QueryProcessor.h"

std::string JsonSki::JSONSkiParser(std :: string query, std :: string file_paths){
```

```
    char* file_path = const_cast<char*>(file_paths.c_str());
    Record* rec = RecordLoader::loadSingleRecord(file_path);
    if (rec == NULL) {
        cout<<"record loading fails."<<endl;
        return "";
    }
    QueryProcessor processor(query);
    std::string output = processor.runQuery(rec);
    return output;
}

Napi::String JsonSki::JSONSkiParserWrapped(const Napi::CallbackInfo& info) {
  Napi::Env env = info.Env();
  Napi::String first = info[0].As<Napi::String>();
  Napi::String second = info[1].As<Napi::String>();
  std::string returnValue = JsonSki::JSONSkiParser(first.Utf8Value(), second.Utf8Value());
  return Napi::String::New(env, returnValue);
}

Napi::Object JsonSki::Init(Napi::Env env, Napi::Object exports)
{
  exports.Set("JSONSkiParser",Napi::Function::New(env, JsonSki::JSONSkiParserWrapped));
  return exports;
}
```

3.1.6 Bindings.h

```
#include <napi.h>

namespace JsonSki {
 using namespace JsonSki;
   std :: string JSONSkiParser(std :: string query, std :: string file_path);
   Napi::String JSONSkiParserWrapped(const Napi::CallbackInfo& info);
   Napi::Object Init(Napi::Env env, Napi::Object exports);
 }
```

3.1.7 Index.js

```
const jsonski = require('./build/Release/JSONSki.node')

module.exports = jsonski;
```

3.1.8 Details for bindings:

We will essentially wrap each C++ function we want to export in NAPI and add it to the exports object using Init. Input parameters and return values from the Napi namespace should be used in every wrapped function that has to be exported to JS. Every function that is wrapped accepts CallbackInfo as an input parameter. We need to inform our node-gyp that we have added more C++ files by using the init function to simply set the export key with a corresponding wrapped function. Index.js contains the Node interface for loading the module and running it using Node.js

3.19 Steps to run after changing c++ files:

- npm run build
- node index.js

3.2 NPM Library [17]

Each JavaScript package in the public npm repository is a collection of software and information. The npm registry is used by both open source developers and corporate developers to get packages for their own projects and to donate packages to the community at large or to other members of their organizations. The steps for publishing are as follows:

- npm run build
- npm login
- npm publish

Refer: https://docs.npmjs.com/about-the-public-npm-registry

3.3 VS Code Extension [16]

V8 engine is used by VS Code Extensions (code.visualstudio.com). The primary framework that enables VS Code for desktop to function across all of our supported platforms is called Electron (Windows, macOS, and Linux). To create cross-platform desktop apps, it integrates Chromium with browser APIs, the V8 JavaScript engine, Node.js APIs, and platform integration APIs. The VSCode addon for JSONSki mostly uses the npm package with built-in VSCode capability APIs to allow user interactivity. The intended walkthrough for using the extension is as follows.

- Open a JSON file in the active Editor and Type JSONSKI.
- Open the Input box (CMD + SHIFT + P) on your VSCode editor.
- Choose between creating a boiler Plate and Querying JSON.
- Enter the JSONSki Query
- Find your results in results.json

3.3.1 Implementation:

```
const vscode = require('vscode');

const fs = require('fs');

const path = require('path');

const JSki = require('jsonski')

import { window } from 'vscode';

function activate(context) {

const vscodeFunctions = {

    openTextDocument: vscode.workspace.openTextDocument,

    showTextDocument: vscode.window.showTextDocument,

    showErrorMessage: vscode.window.showErrorMessage,

    showInformationMessage: vscode.window.showInformationMessage,

    showInputBox: vscode.window.showInputBox,

    getConfiguration: vscode.workspace.getConfiguration,

    showQuickPick: vscode.window.showQuickPick

  };
```

```
        context.subscriptions.push(vscode.commands.registerCommand('jsonski.jsonski',
function () {

                const htmlContent = `const JSki = require('jsonski');

const fs = require('fs');

console.time();

console.log('JsonSki Runtime', JSki.JSONSkiParser("$[*].entities.urls[*].url",
"dataset/twitter_sample_large_record.json"));

console.timeEnd();

file_contents = fs.readFileSync('dataset/twitter_sample_large_record.json')

str = file_contents.toString()

console.log("Javascript Runtime")

console.time();

var json = JSON.parse(str);

console.timeEnd();`;

vscode.window.showInputBox({

    validateInput: text => {

     vscode.window.showInformationMessage(`Validating: ${text}`);

    }});


                const folderPath = vscode.workspace.workspaceFolders[0].uri

                        .toString()

                        .split(':')[1];
```

```
                        fs.writeFile(path.join(folderPath, 'index.js'), htmlContent, (err) => {
                            if (err) {
                                return vscode.window.showErrorMessage(
                                    'Failed to create a boilerplate file!'
                                );
                            }
                            vscode.window.showInformationMessage('Created boilerplate files');
                        });
                        vscode.window.showInformationMessage('Hello World from jsonski npm!');
    }));
}
function deactivate() {}
module.exports = {
    activate,
    deactivate
}
```

Chapter 4

Performance Results

4.1 Benchmarking

JSONSki finds its competitors in libraries like SimdJSON and RapidJSON. Further, it was also essential to compare the execution time against Javascript. Hence, to establish performance for JSONSki it is necessary to benchmark the performance results against these tools. Separate Benchmarking repositories were developed so that users could verify the results locally on their machine. The querying process is segregated into 2 parts:

1. Small JSON files - (0.472 Kb - 561Kb)
2. Large JSON files - (26 Mb - 189 Mb)

Further, Complex queries are also benchmarked to ensure the performance is consistent while querying nested JSON structures.

The performance results were generated using the following queries on the respective files.

Query	File Name	Size
$.brewing.country.id	datasets/small.json	472 Bytes
$.tiger	datasets/mid-pokemon.json	24 Kb
$.koko	datasets/half-pokemon.json	85 Kb
$.tiger	datasets/payload.json	561 Kb
$.sol2.tiger	datasets/test.json	26 Mb
$.tiger	datasets/citylots.json	189 Mb

4.2 JSONSki (C++)

4.2.1 Methods in Comparison

- simdjson-dom: latest C++ version [https://github.com/simdjson/simdjson] using a dom::parser
- simdjson-ondemand: latest C++ version using a ondemand parser
- JSONSki: original C++ version [https://github.com/AutomataLab/JSONSki]
- simdjson_nodejs: [https://github.com/luizperes/simdjson_nodejs] using lazyParse()
- simdjson_nodejs_ours: simdjson ported by us with NAPI
- JSONSki_nodejs: [https://github.com/AutomataLab/JSONSki_nodejs] ported by us with NAPI
- JSON.parse(): Node.js built-in JSON parser, then querying with "." notation

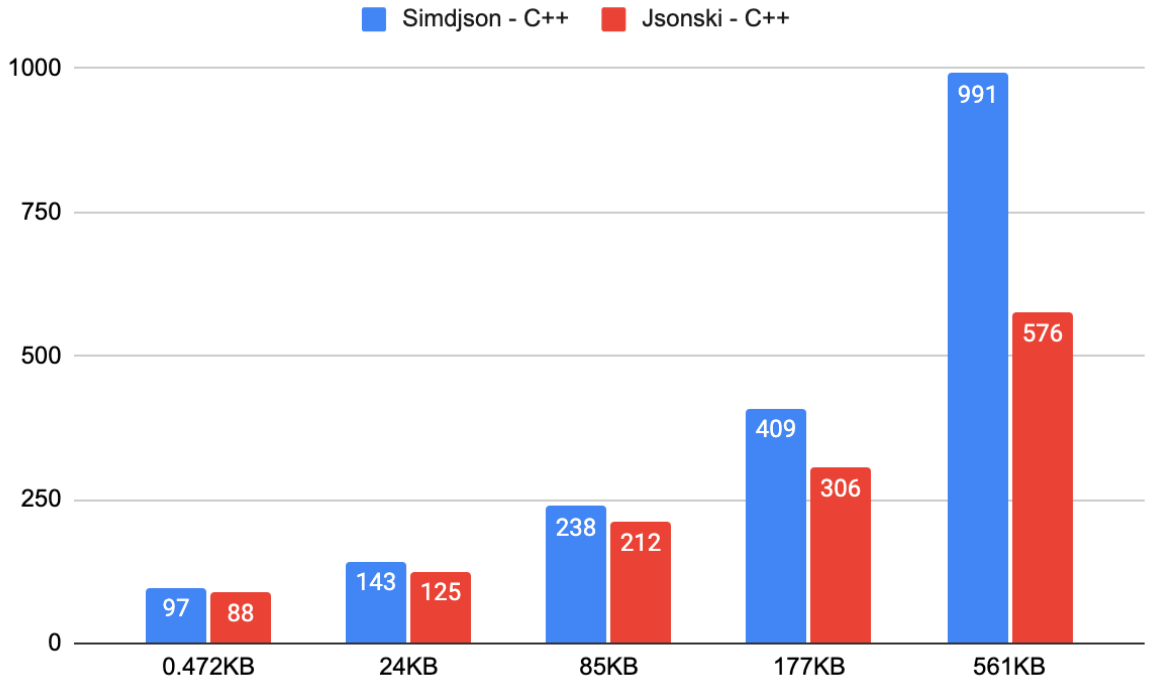
4.2.2 Native Performance Breakdown (chrono time API, excluding first run)

	JSONSki - C++ (load + stream)	simdjson-dom (load + parse + query)	simdjson-dom (load + parse)	simdjson-ondemand (load + parse + query)
472 Bytes	88 µs	97 µs	74 µs	121 µs
24 KB	125 µs	143 µs	123 µs	1020 µs
85 KB	212 µs	238 µs	215 µs	3293 µs
177 KB	306 µs	409 µs	348 µs	6645 µs
561 KB	576 µs	991 µs	936 µs	21284 µs
189.8 MB	213184 µs	541545 µs	494338 µs	734381 µs

4.2.3 Execution time comparison in microseconds

SIZE	Jsonski - C++	Simdjson - C++
0.472KB	88	97
24KB	125	143

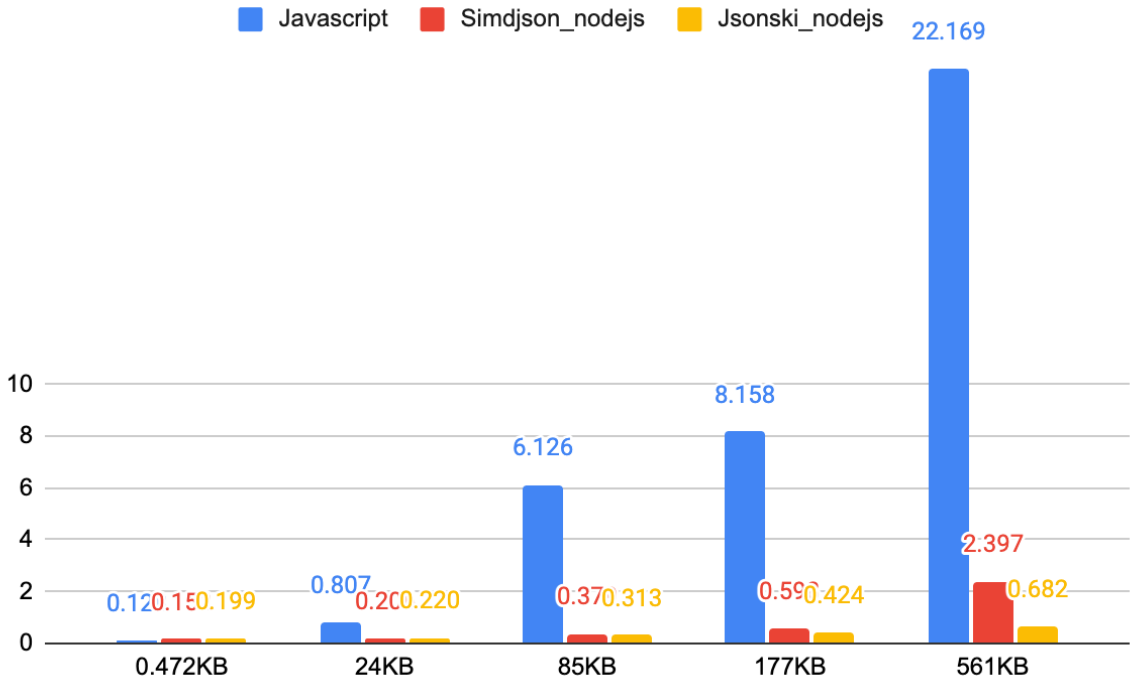
85KB	212	238
177KB	306	409
561KB	576	991
189.8Mb	213184	541545



4.3 JSONSki (Javascript)

	JSON.parse()	simdjson_nodejs	JSONSki_nodejs
472 Bytes	0.128ms	0.152ms	0.199ms
24 KB	0.807ms	0.206ms	0.220ms
85 KB	6.126ms	0.379ms	0.313ms
177 KB	8.158ms	0.598ms	0.424ms
561 KB	22.169ms	2.397ms	0.682ms
26 Mb	359.974ms	122.651ms	14.579ms
189.8 Mb	3133.188ms	927.448ms	180.056ms

NPM Performance Comparisons (JS console.time(), benchmarkify.js, excluding first run)



4.3.1 Benchmarking APIs:

- **Javascript:**
  - Benchmarkify.js

    Refer: https://www.npmjs.com/package/benchmarkify
  - console.time()

    Refer: https://developer.mozilla.org/en-US/docs/Web/API/console/time
- **C++:**
  - <chrono>

    Refer: https://en.cppreference.com/w/cpp/chrono

- **Repositories:**
  - https://github.com/AutomataLab/NPM-JSON-Parser-Benchmarking
  - https://github.com/AutomataLab/JSON-Parser-Benchmarking

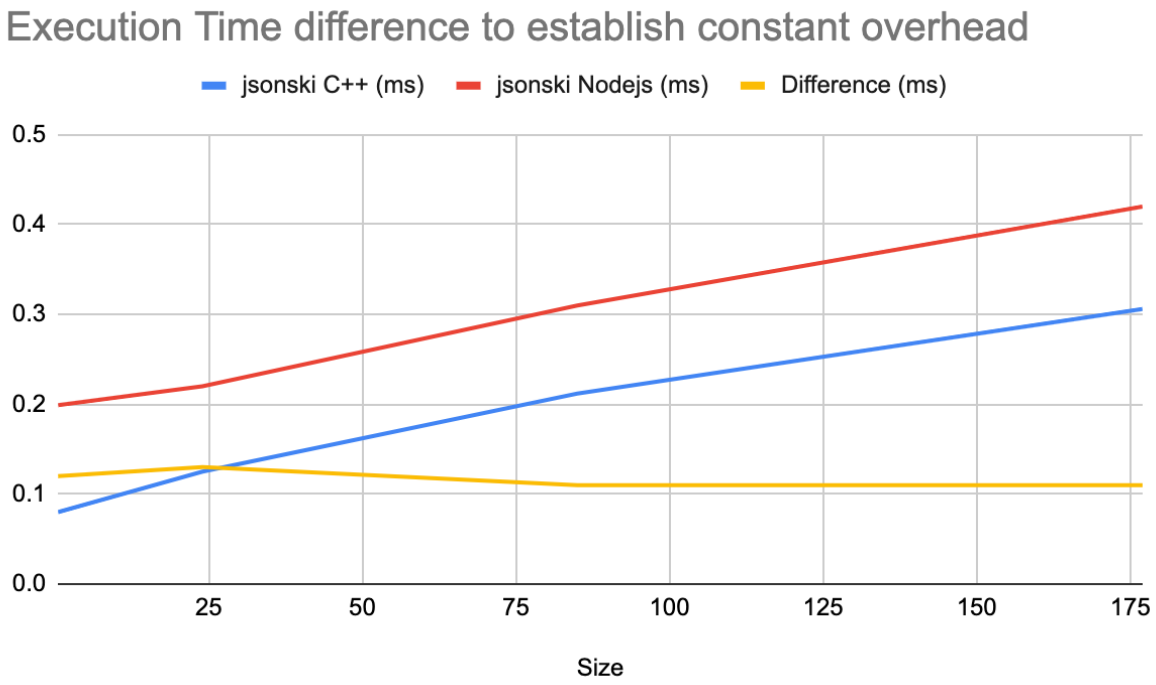
4.3.2 Observations:

- The results of the on-demand parser in simdjson are not consistent with simdjson_nodejs as for smaller data the simdjson_nodejs api has much better performance.
- Majority of the overhead is caused by loading and parsing, as very less part is required for querying and displaying the result.
- The results of dom::parser are better and much more consistent
- Jsonski C++ sometimes performs slightly slower for mid to large sized data, suspecting the reason to be that wrapping cost becomes negligible with increasing size.
- Both lazyParse() and parse make use of the dom::parser API and not the on demand parser
- for a JSON size > 100kb, JSONSki starts to outperform both SIMDJSON and Javascript parsing
- For a JSON size < 100 Kb, Javascript outperforms both SIMDJSON and JSONSki, as size grows beyond, JSONSki becomes exponentially faster.
- Most of the overhead is caused by the wrapping between Javascript and C++ i.e Sending parameters to C++ and receiving a string on Javascript end - this was determined by benchmarking within C++ code.
- Some overhead, although negligible ~0.01ms in comparison to the previous aspect, is caused by type casting String -> char * while managing input parameters.

4.3.3 Wrapping Overhead:

The following experiment establishes that there is a fixed overhead involved in passing the arguments (File Path, Query) to the C++ library and back to the Node.js environment. However, there will definitely be an extra overhead taken to write the data to a file or displaying it as output as the size of the output grows.The following breakdown establishes a fixed overhead caused by Wrapping functions since the only parameters that are passed to native Code are file path and query that could be considered as constants. However, it can be argued that for larger outputs the overhead could exceed since a string is passed back to the Javascript code. Although, it largely depends on the scope of execution. For example, we can write the output to a file through the C++ part of the code and finish the execution process there.

Size	jsonski C++ (ms)	jsonski Nodejs (ms)	Difference (ms)
0.472 Kb	0.08	0.199	0.12
24 Kb	0.125	0.22	0.13
85 Kb	0.212	0.31	0.11
177 Kb	0.306	0.42	0.11



Query	Execution time (ms)	Number of rows
$.features[0:3]	175.94	3
$.features[0:300]	171.23	300
$.features[0:3000]	177.2	3000

Chapter 5

CI/CD and Scalability enhancements

5.1 CircleCI [18]

DevOps techniques can be applied using CircleCI, a platform for continuous integration and delivery. Currently, the project is being supported by:

- CircleCI pipeline for MacOS and linux for JSONSki C++

```
version: 2.1
orbs:
  win: circleci/windows@2.2.0
jobs:
  build-for-linux:
    docker:
      - image: cimg/node:14.10.1
  build-for-macos:
    docker:
      - image: conanio/gcc10
        environment:
          CXX: g++-10
          CC: gcc-10
          CMAKE_BUILD_FLAGS:
          CTEST_FLAGS: --output-on-failure
```

- CircleCI pipeline for MacOS for JSONSki Node.js

```
version: 2.1
orbs:
 node: circleci/node@5.1.0
 python: circleci/python@2.1.1
workflows:
 version: 2
jobs:
 build:
  macos:
    xcode: 14.2.0
```

5.2 Documentation

- Added Doxygen Documentation to JSONSki Repositories
- Added shield IO badges to track npm downloads, CI failures, support and to improve visibility.
- Setup Readme.md for the JSONSki VSCode extension, JSONSki npm library and JSONSki C++ library.

Chapter 6

Conclusion & Future Scope

Data Streaming performs better only when Parsing isn't already performed and we don't have the parse tree in the memory. For example: streaming is useful only when we are sure about making a fixed number of queries and not for cases where we need to query multiple times over a stored parsed object. Replacing JSON.Parse() where the application needs a limited number of queries over an object could potentially use JSONSki. Some of the applications that could use the tool are Postman, CouchDB etc.

Chapter 7

Acknowledgements and References

Acknowledgement

I would express my deep sense of gratitude towards professor Zhijia Zhao for his valuable help and guidance. I am immensely thankful for this opportunity that allowed me to contribute towards his research. This project helped me learn and develop my technical abilities to a great extent owing to the limited documentation and hurdles that fell in my way. I feel I have grown a lot as an individual and the project enhanced my ability to research and grasp new things. I take this opportunity to express my indebtedness towards professor Zhao for this technically challenging yet immensely interesting project.

References

1. JSONSki: streaming semi-structured data with bit-parallel fast-forwarding Lin Jiang , Zhijia Zhao, https://dl.acm.org/doi/pdf/10.1145/3503222.3507719
2. https://nodeaddons.com/getting-your-c-to-the-web-with-node-js/
3. https://stackoverflow.com/questions/54740947/node-js-addons-nan-vs-n-api
4. https://medium.com/jspoint/a-simple-guide-to-load-c-c-code-into-node-js-javascript-applications-3fcccf54fd32
5. https://stackoverflow.com/questions/18382957/tree-parser-vs-stream-parser
6. https://www.slideshare.net/nasottola/next-generation-napi
7. https://medium.com/jspoint/a-simple-guide-to-load-c-c-code-into-node-js-javascript-applications-3fcccf54fd32
8. https://blog.atulr.com/node-addon-guide/
9. https://ducmanhphan.github.io/2018-09-19-Configure-Binding.gyp-file-in-C++-Addon-Node.js/
10. https://github.com/luizperes/simdjson_nodejs/blob/master/binding.gyp
11. https://docs.couchdb.org/en/3.2.2-docs/api/basics.html

12. https://github.com/V0ldek/rsonpath/blob/main/rsonpath-benchmarks/benches/rsonpath_vs_jsonski.rs
13. https://labs.tadigital.com/index.php/2020/06/05/n-api-and-its-benefits/
14. https://medium.com/netscape/javascript-c-modern-ways-to-use-c-in-javascript-projects-a19003c5a9ff
15. https://medium.com/jspoint/a-simple-guide-to-load-c-c-code-into-node-js-javascript-applications-3fcccf54fd32
16. Code.visualstudio.com
17. https://docs.npmjs.com/about-the-public-npm-registry
18. https://circleci.com/