GpJSON: High-performance JSON Data Processing on GPUs

Sacheendra Talluri
Vrije Universiteit Amsterdam
s.talluri@vu.nl

Guido Walter Di Donato
Luca Danelutti
Politecnico di Milano
firstname.lastname@polimi.it

Koen R. Vlaswinkel
Eindhoven University of Technology
k.r.vlaswinkel@student.tue.nl

Marco Arnaboldi
Arnaud Delamare
Oracle Labs, Zurich
{marco.arnaboldi,arnaud.d.delamare}@oracle.com

Marco D. Santambrogio
Politecnico di Milano
marco.santambrogio@polimi.it

Daniele Bonetta
Vrije Universiteit Amsterdam
d.bonetta@vu.nl

ABSTRACT

The JavaScript Object Notation (JSON) format is ubiquitous, and countless applications depend on it to store and exchange high volumes of data. Despite its great popularity, JSON is nevertheless a very inefficient data format: decoding and querying JSON data is often a major bottleneck for many data-intensive applications.

In this paper, we explore how Graphics Processing Units (GPUs) can be used to parallelize *both* JSON de-serialization and querying. We show how JSON parsing can be implemented on GPUs by means of parallel structural index construction, and we describe how JSON data can then be queried *in situ* using a lightweight query engine designed to run on GPUs. We present the design and implementation of GpJSON, a GPU-based JSON data processing library. The library can be used from high-level languages such as JavaScript or Python, and features bindings for the GraalVM language runtime. Our evaluation on real-world datasets shows that, on a single NVIDIA Ampere A100, GpJSON achieves at least 2.9× speedup on end-to-end performance (de-serialization plus querying) over state-of-the-art parallel JSON parsers and query engines, and 6-8× over NVIDIA RAPIDS.

PVLDB Reference Format:
Sacheendra Talluri, Guido Walter Di Donato, Luca Danelutti, Koen R. Vlaswinkel, Marco Arnaboldi, Arnaud Delamare, Marco D. Santambrogio, and Daniele Bonetta. GpJSON: High-performance JSON Data Processing on GPUs. PVLDB, 18(9): 3216 - 3229, 2025.
doi:10.14778/3746405.3746439

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/gpjson-vldb/gpjson.

1 INTRODUCTION

Motivation. JSON [5] is arguably the most popular textual data format on the planet, and the amount of data directly available in JSON is enormous [47]. All major Web platforms and services (e.g., Twitter/X, Facebook or Amazon), continuously produce massive amounts of JSON data through their open APIs [6, 23, 50], and JSON is natively supported by many database systems such as Oracle, MySQL, or PostgreSQL [35, 41]. Unfortunately, analyzing and processing large amounts of JSON data is an expensive task. In particular, studies have shown that JSON parsing alone can take up to 80% of the total time needed to process a single file [38]. Moreover, after de-serialization, data is typically analysed using a query language or hand-written code, which can introduce additional bottlenecks. In this context, new techniques are required to accelerate parsing and querying of JSON documents.

Existing approaches. High-performance JSON parsing and querying is non-trivial [40], as it requires contextual information about the data being parsed (e.g., to escape special characters and identify nested values). Parallel data processing is a viable solution to optimize JSON data access. State-of-the-art high-performance solutions leverage SIMD parallelism [24, 30, 31] or multi-threading [24] to speed up sequential parsing and querying.

The increasing availability of massively parallel GPU architectures on commodity hardware offers even more opportunities than SIMD or multi-threading to achieve high performance. However, leveraging GPUs for parallel data processing is challenging, especially with formats like JSON that were designed for *sequential* data access. Previous research has shown the benefits of leveraging GPUs to speed up general-purpose databases [3, 20, 44], and several commercial GPU-optimized databases have been proposed over the last years, such as HeavyDB [21] and SQreamDB [45]. Such systems do not allow querying data in textual JSON format, and typically use built-in storage or binary formats like Apache Parquet [14] or Apache Arrow [12]. GPUs have already been used to speed up processing of large XML and CSV files [29, 43, 46]. However, to the best of our knowledge, GPU-based solutions for *both* JSON parsing and query processing have not been proposed before.

Contributions. We propose GpJSON, a library for processing JSON data on GPUs. GpJSON uses structural indexes and a lightweight query engine to execute JSONPath queries over LD-JSON documents. The main contributions of this paper are:

- We design and show how structural indexes [31] can be constructed on the GPU (Section 3.1). Structural indexes are auxiliary data structures commonly used by SIMD-based JSON parsers to parallelize data de-serialization, and have never been used in GPU-based data processing systems, to the best of our knowledge.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746439

- We design a new technique to execute queries expressed using the JSONPath [16] query language on the GPU (Section 3.2). Our technique relies on a lightweight query engine designed to operate entirely on the GPU. In our approach, JSONPath queries are compiled to a custom bytecode format and executed on the GPU by means of a bytecode interpreter. To the best of our knowledge, our work is the first to show query compilation targeting a bytecode interpreter operating on a GPU.
- We implement our new techniques in a library called GpJSON (Section 3.2.2). The library uses CUDA and has bindings to dynamic languages such as Python and JavaScript via the GraalVM [10, 52] runtime. In this paper, we use JavaScript as the main language to interact with the library, but GpJSON can also be used standalone or by any other programming language using conventional foreign-function interfaces. GpJSON is open source and is available on GitHub [1]

Evaluation. We conduct an extensive performance evaluation using different GPU architectures. We compare GpJSON against state-of-the-art SIMD-based parsing libraries, popular frameworks for languages such as JavaScript, and the GPU data-processing library RAPIDS cuDF (Section 4). We evaluate the performance of our library using JSON datasets, and compare our performance against alternative solutions on a selection of JSONPath queries. Overall, our implementation outperforms CPU SIMD/multithreading-based and other GPU-based solutions.

2 BACKGROUND

This section introduces JSON and the JSONPath query language. Then, we introduce structural indexes, discussing how existing techniques differ from our GPU-based solution.

2.1 JSON and the JSONPath Query Language

JSON [5] is specified by ECMA-404 and is a relatively simple, text-based data-interchange format. JSON has four primitive types (strings, numbers, boolean and `null`) that can be combined with composed types (arrays and objects). JSON strings are sequences of characters enclosed by double quotes with some escape sequences (e.g. `"\n"` for a newline character). Given the vast amount of available JSON data, JSON objects are very often stored in a single file, where each line corresponds to a distinct value. Such files are called *Line Delimited* JSON documents (LD-JSON), and are the main data format supported by GpJSON (as well as by many popular systems, e.g. [13]). Querying JSON objects often involves selecting and filtering data from LD-JSON documents. One of the most common ways to query such documents is by using JSONPath [16]. JSONpath is an expression language used to filter and transform JSON documents inspired by the XPath language for the XML format. The following is an example query:

```
$.users[?(@.lang == "en")]
```

which selects all users in a LD-JSON document whose language is `"en"`. JSONPath is supported in many domains and applications (e.g., Kubernetes [28], Oracle RDBMS [9]), and it is often employed in Extract-Transform-Load (ETL) workloads, where it is typically used to extract data from large JSON documents. GpJSON uses JSONPath as its query language.

[1] https://github.com/gpjson-vldb/gpjson

2.2 Structural Indexes for JSON Analytics

Broadly speaking, structural indexes are data structures used to accelerate data lookups based on some metadata. In the context of JSON analytics, they are often used to track the location of certain symbols (e.g., all `"{"` characters). Structural indexes can also be used to represent more complex structures, and they can be extended to encompass data on multiple levels. So-called *multi-leveled* structural indexes can be used to store data about nested data structures, such as tree-like objects, or nested JSON objects and arrays.

Popular JSON processing libraries often store structural indexes in *bitmap indexes*, using only one bit to represent one byte of the input data, allowing a compression ratio of 8 to be achieved without efforts. One of the advantages of such (bitmap) structural indexes is that they facilitate SIMD processing in several ways. For example, depending on the supported instruction set, SIMD instructions can execute AND operations on 128, 256, or 512 bits at once, in comparison to the usual 8, 16, 32, or 64 bits of a single instruction on a single integer type.

In the context of JSON analytics, Mison [31] was one of the first systems to use SIMD to process JSON data. It functions by scanning a document several times, discovering different structural elements (such as `"{"` and `","` at every scan). Using these structural elements, it can then quickly locate values in large documents. Mison utilizes multiple structural indexes that encode the usage of specific structural characters. This approach would be simple if these characters appeared solely as structural elements. However, they can also be found within string literals. It is therefore necessary to first determine whether a particular character is part of a string. Since characters can also be escaped within strings, this is not trivial. For example, the string literal `\"foo\"bar"` only ends at the last quote character, so it is crucial to determine which quote characters are escaped. It is impossible to do this by only looking at the previous character of a quote character since the escape backslash can also be escaped itself, so it is necessary to fully understand the context in which the quote character is used. To this end, systems like Mison use multiple bitmap indexes and bit-parallel bitmap SIMD instructions to create a structural index that contains for every character whether it is contained within a string. Here is an example of such an index:

```
{"foo": { "object": "bar" }}
00111100000111111100011110000
```

This bitmap can be used to construct a leveled-colon bitmap, which contains whether a specific colon character denotes a comma within the level. A *level* is defined as the depth within a JSON record, and as such, the root object is defined as level 0 (L0), an object contained within the root object as level 1 (L1), etc.. Here is an example of the final structural index used by Mison to query data:

```
   {"foo": { "object": "bar" }}
L0 0000001000000000000000000000
L1 0000000000000000010000000000
```

Structural indexes similar to the ones used by Mison are employed by other systems with similar goals (e.g., `simdjson` [30]

and Pison [24]). Our implementation of a GPU-based JSON parsing library also relies on similar structural indexes. Unlike existing approaches, however, all indexes used by GpJSON are stored in the GPU memory, and are constructed and manipulated from GPU kernels. As we will discuss, creating and accessing such indexes from GPUs presents unique challenges (and opportunities) that makes it impossible to simply port existing SIMD-based approaches to the GPU.

3 GPJSON DESIGN AND IMPLEMENTATION

Executing JSONPath queries on a GPU is attractive because CPU-based approaches struggle with latency and throughput when processing large amounts of JSON data. Existing JSONPath libraries often rely on tree-walking interpreters or recursive-descent parsers, which are inherently sequential and suffer from cache inefficiencies due to JSON's irregular structure. In contrast, a GPU-based approach can exploit massive parallelism to evaluate multiple JSON objects simultaneously. Furthermore, GPUs offer high memory bandwidth, making them well-suited for workloads involving large JSON datasets. However, the challenge lies in the heterogeneous and hierarchical nature of JSON, which introduces memory divergence, irregular control flow, and varying object sizes, requiring efficient utilization of GPU resources. GpJSON's design addresses these challenges in two main ways. First, the library adopts a data partitioning approach that allows it to process multiple JSON objects at the same time, thus maximizing parallelism utilization (Section 3.1.1), while reducing the need to perform dynamic memory allocations on the GPU. Second, it performs both JSON parsing and JSONPath query execution on the GPU, thereby avoiding additional memory transfers from DRAM to GPU memory. This is done with multiple GPU kernels, which first construct structural indexes in GPU memory (Section 3.1) and then use them to perform query execution (Section 3.2).

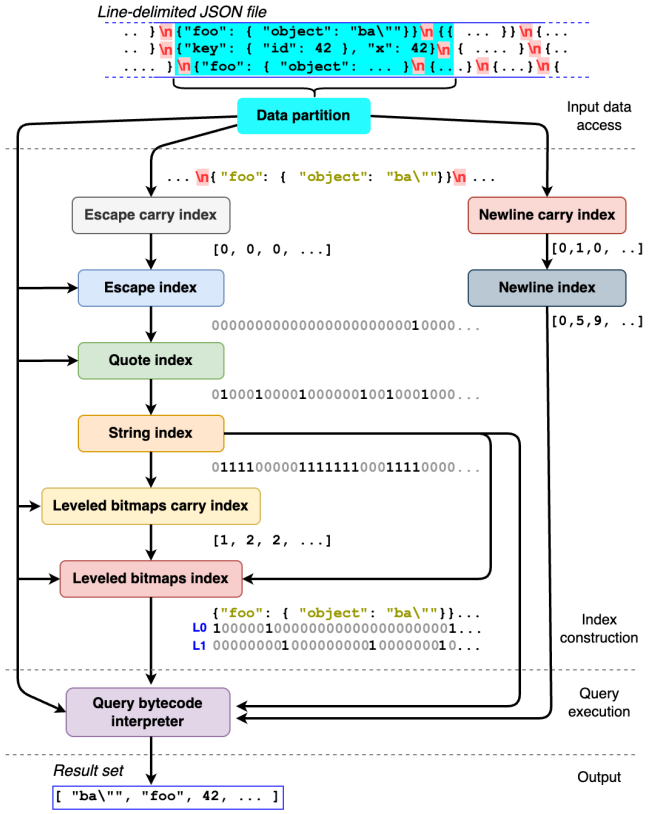

Figure 1: GPU Kernels and indexes in GpJSON. Each index is created by a dedicated kernel. Arrows correspond to data dependencies between kernels.

3.1 On-GPU Structural Indexes Construction

GpJSON relies on the construction of several structural indexes to parse and query JSON data. All indexes are allocated on the GPU memory, and are constructed in parallel using dedicated GPU kernels. Each kernel processes one or more indexes and stores its results in GPU memory. An overview of the kernels used by GpJSON as well as of the data exchanges between them is shown in Figure 1. Data is never copied between GPU kernels, but rather passed by reference, making the entire index construction process zero-copy.

As discussed in Section 2.2, there are multiple variations of structural indexes. simdjson [30] constructs a Tape, while Mison [31] (and Pison [24]) use a Structured Leveled bitmaps index. GpJSON uses a variation of such Leveled bitmaps index, optimized for GPU construction and storage. The following figure provides an example of the index:

```
    {"foo": ["alpha", "beta"], "gamma": {"lambda": "kappa"}}
L0  1000001000000000000000000100000000100000000000000000000001
L1  0000000010000000100000001000000000001000000001000000001000
```

As can be seen in Figure 1, GpJSON performs several intermediate steps before constructing the Leveled bitmap index. Once created, the index is then used by GpJSON to execute JSONPath queries. GpJSON's Leveled bitmaps index differs from those used in Mison (and Pison [24]) in several ways. First, we do not build different bitmaps for different characters initially. This reduces the amount of memory required (by approximately 20% in the best case). Memory is usually more constrained on a GPU device than on a host device due to less expandability and a fixed amount of memory for a specific device model, so this brings tangible benefits. Second, our final Leveled index contains all structural characters, including characters that start or end a level ("{", "[", "}", and "]"). Other approaches [31] only save the positions of colons and, in the case array indexes are part of a query, commas. By also saving the position of other structural characters, we can only use one index (instead of two indexes used in [31]), which again reduces the amount of memory (in this case by approximately 10% in the best case compared to storing two different indexes). This also allows us to determine the end of a level more quickly since it is unnecessary to iterate over two indexes to find it. Other frameworks (e.g. [24]) build on the structural Leveled colons bitmap index proposed by Mison, and use SIMD instructions to speed up their construction. Such SIMD techniques are not applicable in the GPU-based SIMT model, since CUDA does not have SIMD instructions that operate

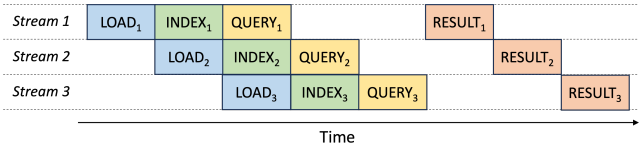

Figure 2: Pipelining of data access, index creation and query execution for different data partitions.

on vectors more than 64 bits, while Pison uses 256-bit SIMD instructions and uses operations that are unavailable for CUDA (such as, e.g., `pclmulqdq` carry-less multiplication).

3.1.1 Data Loading and GPU Memory Management.

Another notable difference between SIMD-based techniques and GpJSON is about data loading and memory management. GPU memory is typically smaller than DRAM, and storing a full file in it may not always be possible. GpJSON addresses this limitation by splitting the input file into equally-sized partitions and processing them concurrently with multiple GPU streams [36]. The partition size is adjustable based on the GPU memory size.

Input data partitioning brings two main advantages. First, it allows handling files bigger than the GPU memory by letting GpJSON pipeline data loading, index construction, and query execution, thus enhancing efficiency as shown in Figure 2. A second key benefit of splitting the input data into uniformly-sized partitions is that most structural indices for JSON parsing can maintain a constant size (relative to the partition size) and be pre-allocated on GPU memory. Consider the example indexes in Figure 1. Knowing the partition size enables pre-allocation of the *Escape*, *String*, and *Quote* indexes, each needing 1/8th of the partition (one bit per byte). Similarly, the *Leveled bitmap* index can be pre-allocated, as its size is proportional to the JSONPath query's maximum depth and partition size. Indexes like the *Newline* index must be dynamically allocated since the GPU needs to first identify the number of new lines in a partition before storing their offsets. However, as most indexes are of fixed size and pre-allocated, the effect of dynamic memory allocation during JSON parsing is negligible, as evaluated in Section 4.4. After executing a query on a partition and collecting the results, a new partition can be loaded, and its indexes can be built using the same pre-allocated data structures already available in GPU memory.

3.1.2 Newline index. Before GpJSON can build (and query) leveled bitmaps indexes, it first needs to identify all single JSON objects in a LD-JSON file (i.e., identify all new lines). To this end, GpJSON builds a first index containing the offsets for each new line character (`"\n"`) in a file. Each entry in such *Newline* index will then have a corresponding Leveled bitmaps index.

Building a parallel Newline index is non-trivial on a GPU, since the number of newlines within a file is variable and highly dependent on the input data. Therefore, it is unknown how many newlines there are in the file, and as such, it is unknown how much memory is required to store the complete newline index. One of the challenges of the GPU is that it is not possible to allocate large amounts of memory within a kernel dynamically. Therefore, the host device needs to allocate the memory beforehand to make sure there is a contiguous block of memory that can be used in later steps. This limitation has been solved through the use of two separate (but dependent) kernels.

The first kernel scans the number of newlines in each thread and stores that information in a block of memory that is dependent on the number of threads running the kernel rather than on the size of the input data. The number of kernels is constant, so the required memory can be pre-allocated. The detected newlines will then be used to compute the total amount of memory required to store the Newline index. This amount can be computed by simply summing all of the values and multiplying it by the amount of memory required for every position. Since only the numerical position is stored, this would usually be an 8-byte number. Thus, the amount of memory required to store the Newline index can be computed as in eq. (1). Once the required amount of memory is known, this amount should be allocated on the GPU, such that it can be used to store the index.

$$\text{newline index size} = 8 \cdot \sum_{i=1}^{\text{thread count}} \text{counts}[i] \tag{1}$$

The second kernel will store the actual positions of all newlines in the Newline index. One of the most important aspects is ensuring that the newlines are in non-overlapping places in the index. One of the other aspects is making sure they are in ascending order. While the order is mostly irrelevant to querying the values, it is important to ensure results can be correlated back to lines. To ensure that every GPU thread can place their discovered newlines in the correct position in the Newline index, the previously discovered count of newlines is used. This is used to create an array of offsets, which will determine where each thread can start writing its discovered newline positions. For example, if there are 4 threads and the result is count $= \{1, 6, 3, 1\}$, then the offsets would be offsets $= \{0, 1, 7, 10\}$. In this case, the size of the Newline index would be 11 elements, resulting in a memory size of 88 bytes.

3.1.3 String index. After all new lines in a LD-JSON document have been identified, GpJSON analyzes each JSON object. The goal is to ensure that all structural characters (such as, e.g., `"{"` or `"]"`) are ignored if and when they appear within a string. To this end, a second index called the *String index* is used. For every character of a JSON object, the index will indicate whether it is part of a string literal. It will do so using a bitmap index, and as such, it is only an eighth of the size of the input data. The construction of the String index cannot be performed in a single pass, because the GPU needs contextual data about text fragments being processed in other parallel threads. GpJSON therefore relies on other intermediate (temporary) indexes to facilitate the construction of the index. Such indexes are, namely: an Escape carry index, an Escape index, a Quote index, and finally the String index, as shown in Figure 1.

First, the Escape carry index needs to be built. This index is similar to the first step of the Newline index, although in this case, it is used to resolve dependencies between kernels. This index essentially computes whether there is a "carry" of the escape, i.e., whether the first character of the next GPU thread needs to be escaped. This only works when we assume that there are no runs of escapes that run for the kernel duration since that would introduce dependencies between these kernels. However, the input data of one kernel is always at least 64 bytes, and as such, it would already be improbable

that all 64 bytes are escape characters. If this does happen, this can easily be detected by counting the number of escapes compared to the total number of bytes processed by the kernel. To compute the Carry index, it is enough to start with an initial carry of False, then looping over all the characters processed by the kernel once again. When the character is a backslash character ("\"), the carry needs to be XOR'ed with True. If it is not a backslash character, the carry is reset to False. All such computations are implemented in GpJSON within GPU kernels, without ever involving the CPU.

Once the Escape carry index has been built, the Escape index can be computed. The Escape index is similar to the final String index, in that it encodes using a bitmap for each character whether it is escaped or not. Here is an example Escape index:

```
"an e\nsca\nped string\""
0000001000010000000000010
```

Next, the Quote index is created. It is again a bitmap index, this time storing the position of quotes within the file. Here is an example for such index:

```
"an e\nsca\nped string\""
1000000000000000000000001
```

Finally, the String index is constructed from the Quote index and the Quote carry index. The Quote carry index determines whether the final result needs to be inverted, using the value of the $n-1$th kernel. The Quote index is used to compute a prefix-XOR sum, of which the result tells which bytes are in a string literal and which bytes are outside of it. For example, given a value $x = \texttt{01001000}$, it computes the result `01111000`. The value of a specific bit within the result can be computed as in eq. (2). In other words, the prefix-XOR value of a bit i in the result is the XOR of all bits up to and including bit i in input x. Simdjson [30] uses the specialized CPU instruction `pclmulqdq` to accelerate this operation. While there are ways to accelerate the normal computation of this value using GPUs [4], CUDA does not have an instruction to calculate the value instantly and, as such, it is manually implemented in GpJSON's GPU kernels.

$$\textsc{PrefixXORSum}(x, i) = \bigoplus_{k=0}^{i} x[k] \tag{2}$$

It is then XOR'ed with a bitstring of all 1's when the previous string ended in an escape, or not changed if the previous string did not end in a string. This bitstring is then computed for the following string, such that it can be used for the next iteration. Each iteration loops over one 8-byte value, and as such, it processes 64 bytes at once. This kernel does not access the original input data and only requires memory access to the quote index. Here is an example of the final String index as computed after all required steps:

```
{"foo": { "object": "bar" }}
0111100000111111100011110000
```

The pseudocode for the kernel that computes the final String index is shown in Algorithm 1. From all of the created indexes, only the String index is kept in memory to create the final leveled bitmaps index. All other indexes can be discarded since those are only used for the construction of the String index.

3.1.4 Leveled bitmaps index. Once the String index has been determined, the Leveled bitmaps index can be finally computed. The Leveled bitmaps index is a bitmap index once again, but this time

Algorithm 1: String index kernel

Input: The index t of the thread
Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: The quote index q of kernel 3
Input: The quote index carry c of kernel 3 for this thread
Input: The result quote index r

```
1: prev ← 0
2: if c == 1 then
3:     prev ← SetAllBits(prev)
4: for p ← p_s to p_e do
5:     r[p] ← PrefixXORSum(q, p) XOR prev
6:     prev ← r[p] >> 63
```

it stores at which positions structural characters (such as "{", "[", "}", "]", ":" and ",") are located, together with the depth level in the JSON object. For the construction, similarly to the previous indexes, there are two steps; the first step creates a carry index to resolve dependencies between kernels and the second step creates the actual index. An example of a Leveled bitmaps index was presented in Section 2.2, and is also available at the bottom of Figure 1.

The carry index for the leveled bitmaps index contains the level at which the parallel traversal of the file starts, with one value for each traversal performed by a dedicated kernel. To calculate at which level each kernel starts, it is sufficient to count the number of structural open/close characters. The final value of each carry kernel is essentially the number of opening structural characters minus the closing structural characters. Using this, it is possible to calculate the starting level of each kernel for the following step. One important consideration here is that the structural characters could also be located inside strings, so the string index needs to be used to filter out any such character inside strings.

Once the Carry index has been constructed, a process very similar to the offsets calculation of the Newline index is used (section 3.1.2). The previously calculated values are summed, and each intermediate sum is stored in the index. After this index has been constructed, the actual leveled bitmaps index can be created.

The Leveled bitmaps index goes over each character and uses a similar process to the Carry index creation. Characters inside strings are skipped, and structural characters are used to create the actual Leveled bitmaps index. When an opening structural character is discovered, the current level is incremented, while when a closing structural character is discovered, the current level is decremented. When another structural character (":" and ",") is discovered, it is stored at the current level, which also happens for the opening and closing structural characters. Storage is again a bitmap for every character, except that there are multiple levels, so the storage is not just 1/8th the size of the input data but rather depends on the number of levels required for a given query. The maximum number of required levels is computed by GpJSON by (statically) analyzing the JSONPath query before executing it. Any structural characters which are deeper inside the JSON tree structure than what is required for the JSONPath query to be executed are not stored to limit memory usage. The pseudocode for the kernel that computes the final Leveled bitmaps index is shown in Algorithm 2.

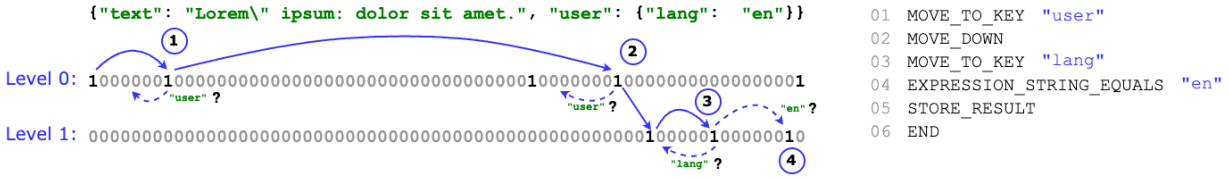

Figure 3: Index traversals during query execution and generated bytecode for the query `$.user[?(@.lang == "en")]`

Algorithm 2 Leveled bitmaps index kernel

Input: The index t of the thread
Input: The starting position p_s of the kernel
Input: The ending position p_e of the kernel
Input: The string index s of kernel 4
Input: The leveled bitmaps carry index c
Input: The result index r
Input: The size of a level n

```
level ← c[t]
for p ← p_s to p_e do
    if not InString(s, p) then
        if IsStructureOpen(p) then
            level ← level + 1
            SetBit(r, n · level + p/64, p mod 64)  ▷ Set the bit p mod 64 in r[n · level + p/64] to 1
        else if IsStructureClose(p) then
            level ← level − 1
            SetBit(r, n · level + p/64, p mod 64)
        else if IsColonOrComma(p) then
            SetBit(r, n · level + p/64, p mod 64)
```

3.2 Querying

Once the Leveled bitmaps index for a JSON object is computed, it can be used to execute a JSONPath query. Query execution is performed in two steps. First, the JSONPath query is compiled into a bytecode-like set of instructions that encapsulates the operations to be performed on the input data and on its corresponding leveled bitmaps index. Then, the bytecode instructions are executed on the GPU to obtain the query results. Compilation of a JSONPath expression to bytecode is achieved following conventional parsing and code generation techniques: the expression is first converted into tokens by means of a lexer that can recognize the JSONPath grammar, then each token is analyzed to create an Abstract Syntax Tree (AST) of the query, which is then traversed to emit bytecode instructions. When the bytecode for a specific query is generated, it can be executed. To this end, GpJSON operates a dedicated GPU kernel that implements an interpreter for the bytecode. Crucially, bytecode instructions correspond to operations to be performed on the leveled bitmap index: some instructions traverse the index to reach a specific position in the input data, while other instructions match values in the data searching for specific tokens. When a match cannot be found, the interpreter terminates, and no results are saved for the given object. When all instructions are executed, a result for the query is found. Figure 3 shows an example of the bytecode instructions generated for the query `$.users[?(@.lang == "en")]`. After a query is compiled to bytecode, it can be executed on the GPU. To this end, each GPU thread receives a subset of all lines to be processed (computed using the Newline index). For each line, a bytecode interpreter executes each instruction on all its assigned lines. First, the beginning of a JSON object is identified using the Newlines index (Figure 3, step 1). Then, the execution of `MOVE_TO_KEY "user"` causes a scan of the Leveled bitmap index to identify the first occurrence of `user` (Figure 3, step 2). If found, GpJSON will then execute `MOVE_DOWN`, which results in the traversal of the index down to level 1 (L1 in Figure 3, step 3). Then, the execution of `MOVE_TO_KEY "lang"` will again trigger a string search until the token can be found (Figure 3, step 4). Finally, executing `EXPRESSION_STRING_EQUALS "en"` will result in a string comparison. Query results are then saved per each line after the bytecode is interpreted on each JSON record.

3.2.1 Bytecode instructions overview. The set of instructions supported by the GpJSON bytecode interpreter can be summarized in the following three categories:
1) Index navigation operations are tree traversal operations that are performed on a structural index to identify relevant values in a given object. They include:

`MOVE_UP`: goes up one level in the Leveled bitmaps index
`MOVE_DOWN`: goes down one level in the bitmaps index
`MOVE_TO_KEY`: moves to a specific key inside an object
`MOVE_TO_INDEX`: moves to a specific index in an array

2) Query execution and control flow operations are operations that drive the state of query execution. They include:

`END`: ends the execution on the current line
`STORE_RESULT`: stores the current result

3) Expression execution operations: are bytecode instructions that can perform computations on the input data. They typically correspond to simple expressions required by the JSONPath language. For example, the `EXPRESSION_STRING_EQUALS` bytecode instruction compares an input string with a constant value.

3.2.2 Implementation and Current Limitations. GpJSON is implemented in CUDA (using GrCUDA [34]) which allows CUDA libraries to be exposed to language runtimes in GraalVM [37]. Using CUDA and GraalVM, GpJSON can be exposed to a variety of different programming languages (from JavaScript to Java and Python) in a straightforward way, but nothing of the GpJSON design is specific to GraalVM. Besides the way GpJSON is exposed to language runtimes, there are two aspects of its implementation that are worth mentioning, namely how GpJSON handles batch processing and how it overlaps computations. In the current implementation, GpJSON supports JSONPath queries for which the

maximum number of results can be known ahead of time. This includes the large majority of queries, with some notable exceptions. Specifically, recursive descent operations like `$..author`, or dynamic array elements queries like `$.user[2:]` (from element 2 until the end of the array) are not supported. GpJSON also lacks support for expressions that allow a user to make assertions on queried values: supporting all such expressions in our implementation would simply mean extending our bytecode instructions set with new instructions mapping to each new language feature, and would not affect the overall design of the system. Finally, the current implementation of GpJSON assumes the input JSON-LD files to be well-formed. This is analogous to existing SIMD-based systems such as Mison [31] or Pison [24], as they all assume the data to be stored in a trusted source (e.g., stored in a Data lake with validation at data ingestion time). Validation could be added to GpJSON by performing a runtime analysis of the indexes discussed in Section 3, analogous to [40].

4 EXPERIMENTAL EVALUATION

In this section, we evaluate our implementation of GpJSON on multiple datasets and queries. We compare GpJSON against several other state-of-the-art JSON parsers and query processing systems.

4.1 Methodology

GpJSON can be accessed from a variety of languages. In this work we focus on the JavaScript language bindings and on the Node.js framework, which feature rich support for JSONPath, as JSON was originally designed for JavaScript. Given that other popular JSON processing libraries are implemented in languages such as C++ or Java, we also compare the performance of GpJSON against such libraries. In all experiments, execution time is largely dominated by JSON data de-serialization and query execution. Therefore, the language runtime is only marginally involved in the total execution time, making it still possible to perform an end-to-end comparison between different approaches. For all the experiments in this section we always measure the total, end-to-end execution time, which includes index construction, query execution, and copying results to the CPU for GpJSON. To shed light on the performance of the main individual index construction steps, we also performed a dedicated set of experiments in Section 4.4.

Another important aspect of the performance evaluation is query execution. While many libraries for JSONPath exist (mostly for JavaScript and Java), the JSONPath language is not directly supported by popular libraries such as simdjson. Given this limitation – and with the goal of still performing a fair comparison – we have implemented the equivalent JSONPath query execution using the library language where appropriate.

We evaluate GpJSON against several libraries, namely: the official **simdjson** [30] C++ library; **Pison** [24], which supports parallel SIMD-based JSONPath queries; **RapidJSON**, a widely used parser by Tencent [48]; **Java JSONPath** [1], which natively supports JSONPath and employs Java Streams for parallel (multi-threaded) query execution; the Node.js-based implementations **Node jsonpath** [7] and **jsonpath-plus** [2], both using the built-in `JSON.parse` function for JSONPath queries; a **manual extraction** approach in Node.js that bypasses library overhead by directly parsing and querying JSON objects; the **Node.js simdjson** bindings for the C++ simdjson library, enabling lazy parsing and efficient JSON access using a dedicated query language. Unfortunately, it is not possible to compare GpJSON directly against Mison [31], given that the system is not publicly available. The implementations that support JSONPath are Java JSONPath, Node jsonpath, Node jsonpath-plus, and GpJSON. We evaluate GpJSON on three different datasets, similar to what was used to evaluate Pison [24]. The datasets are LD-JSON documents containing real-world JSON data. In detail, we used a Best Buy (BB) product dataset, a stream of tweets from the Twitter/X (TT) developer API, and a Walmart (WM) product dataset. Each original dataset is approximately the same size, about 1GB, and – depending on the experiment – we replicate the dataset multiple times to measure GpJSON's performance on multiple data scale factors and larger inputs. We evaluate GpJSON using 6 different JSONPath queries, listed in Table 1. All experiments were conducted on the Oracle Cloud Infrastructure (OCI) on four different types of machines, to evaluate different NVIDIA GPU generations. In detail, the four shapes we used in our tests are:

- ***VM.GPU2.1***: NVIDIA Tesla P100 16GB, 24 cores, Intel Xeon Platinum 8167M (2.0GHz, max frequency 2.4Ghz), 72GB RAM.
- ***VM.GPU3.1***: NVIDIA Tesla V100 16GB, 12 cores, Intel Xeon Platinum 8167M (2.0GHz, max frequency 2.4Ghz), 80GB RAM.
- ***BM.GPU4.8***: NVIDIA Tesla A100 40GB, AMD EPYC 7J13 (2.55 GHz, max frequency 3.7 GHz), 2048GB RAM.
- ***Optimized3 HPC***: 12 cores, Intel Xeon 6354 (3.0GHz, max frequency 3.6GHz), 84GB RAM.

All C++ programs are compiled with the "`-O3`" optimization flag and required instruction set flags (e.g. `-mpclmul` or `-msse4`). The timing results are the average of 10 runs, after 5 warmup runs. By recording the times in the same process in charge of executing the benchmark, errors caused by kernel pauses during communication between processes were reduced. Also, we keep track of the number of results to ensure correctness. All servers run on Ubuntu Server 20.04 and are installed with GraalVM 22.1.0, GrCUDA 0.4.0, G++ 9.4.0, Node.js 18.14.1 (which ships with the V8 JavaScript engine). The GpJSON runtime runs on the GraalVM implementation, while all the other Node.js scripts run on the official Node.js implementation. The Intel CPUs support Intel® SSE4.2, Intel® AVX, Intel® AVX2, and Intel® AVX-512 covering all the instruction sets required by Pison, RapidJSON, and simdjson.

We use GpJSON as the baseline for benchmarking, to highlight any performance differences compared to other JSON parsing and querying implementations. Given that no approaches exist that can process JSONPath queries on a GPU, we first compare GpJSON against existing, non-GPU state-of-the-art techniques. Simdjson is

Table 1: Selected queries

Dataset	Queries	Number of results
TT	`$.user.lang`	150135
TT	`$.user.lang, $.lang`	300270
TT	`$.user.lang[?(@ == "nl")]`	405
TT	`$.user.lang[?(@ == "en")]`	137559
WM	`$.bestMarketplacePrice.price, $.name`	288391
BB	`$.categoryPath[1:3].id`	459332

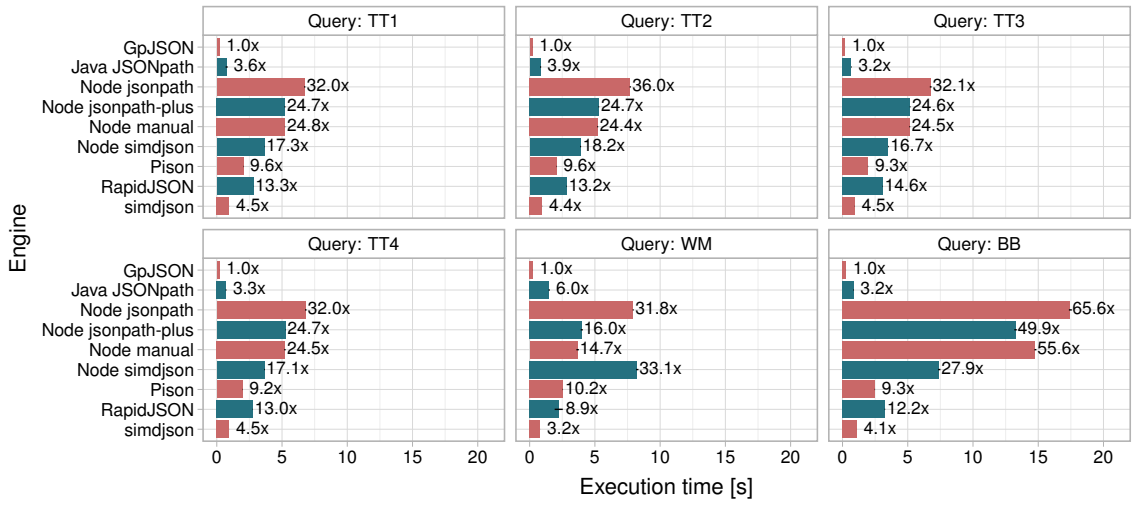

Figure 4: Comparison of GpJSON with other libraries on all queries and datasets. GpJSON is running on A100. Lower is better.

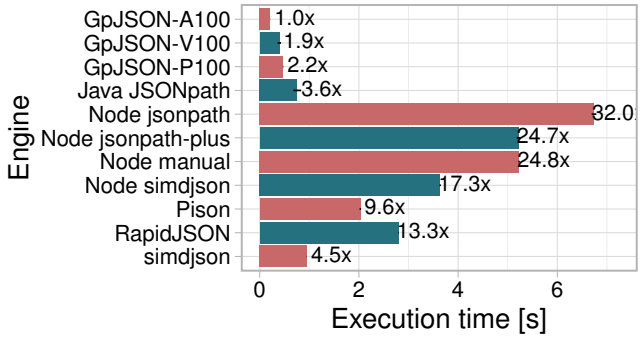

Figure 5: Comparison between GpJSON running on A100 (GPU4.8), V100 (GPU3.1), and P100 (GPU2.1) and all the other libraries running on the HPC shape. This the performance for the Twitter dataset and query TT1. Performance for other queries is similar. Lower is better.

currently the most cutting-edge JSON parsing library, so we will be evaluating it against GpJSON in many of our tests. The usability of GpJSON and simdjson differs significantly because GpJSON is available from a wide range of high-level scripting languages and supports JSONPath, while simdjson is available from C++ through its own API and does not support any high-level query language. Besides SIMD-based JSON processing, we also compare GpJSON against multi-threaded approaches such as Java JSONPath. Since the smallest core count is 12, we have set a fixed thread count for the Java JSONPath experiments on all three machines to ensure a fair comparison. By default, Java streams use *n-1* threads, where *n* is the number of available cores.

4.2 Experimental Results Vs. CPU SIMD

Figure 4 reports the end-to-end time required to query the chosen datasets with the chosen queries. We can see that GpJSON beats all other libraries. First, we can notice that all Node.js implementations take much longer to execute. The performance of our implementation far outperforms that of any Node.js implementation. Node.js manual extraction is the fastest implementation, however, it does not support JSONPath and is manually written to extract the correct values. Due to the overhead of using the library and processing JSONPath, the JSONPath-compatible implementations are much slower. Comparing the plain Node.js implementation (manual) with Node jsonpath and Node jsonpath-plus, it is evident that querying overheads are relatively small, given that the performance is bound by JSON parsing. The Java JSONPath and the C++ implementations are the ones that are closer to GpJSON, with Pison and RapidJSON obtaining very similar results. On the server equipped with the most advanced NVIDIA architecture we have tested, Ampere, we are 3.2× to 4.5× times faster w.r.t. simdjson and 3.2× to 6× times faster w.r.t. Java JSONPath, the two closest implementations to GpJSON in terms of speed. All Node.js libraries struggle to keep up with the other implementations using the Best Buy (BB) dataset, due to the large number of records.

Figure 5 reports a comparison between GpJSON running on GPU shapes and the other libraries running on the HPC shape. The trend is the same, with Node.js solutions slower than the GpJSON and C++ implementations. Our solution (running on an NVIDIA A100) is 3.1× to 4× faster than simdjson and 4.6× to 9.4× faster than Java JSONPath. GpJSON is about 1.8× times faster running on the Ampere generation w.r.t. the Volta one. We don't see the same delta when comparing the Volta GPU with the Pascal one, gaining only about 20% speedup. Note that, regarding CPU libraries, the HPC shape performs about the same as the GPU4.8 one. Thus, from now on, we will mostly compare CPU libraries on these two machines. The performance of Node simdjson shows that there is significant overhead when using simdjson in Node.js due to implementation differences and fewer possible optimizations. Our implementation is at least 15× times faster than Node simdjson, while using the same high-level scripting language.

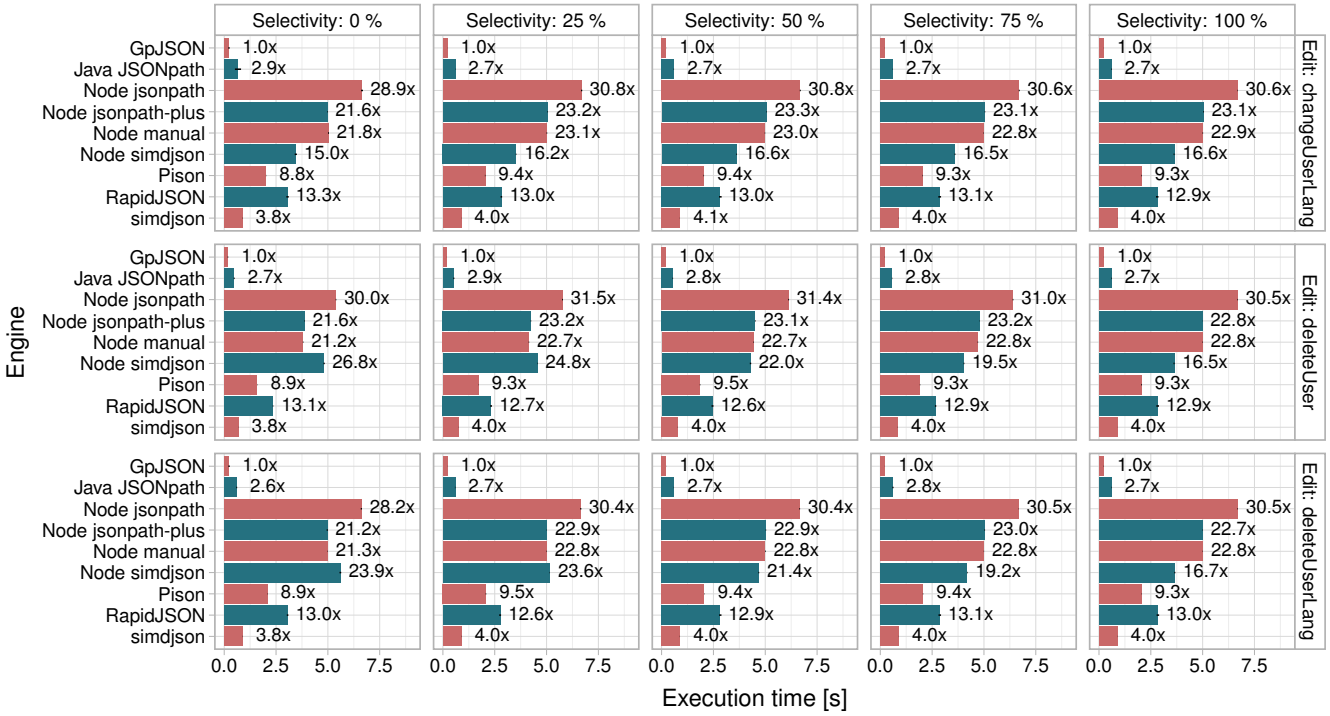

Figure 6: Comparison between GpJSON running on A100 (GPU4.8) and libraries running on the HPC shape. The dataset has been edited in different ways to change the number of matches during query execution. Lower is better.

4.2.1 Varying the number of matches (Selectivity). Next, we analyze the impact of selectivity, which we evaluated by keeping the same dataset size but removing the `$.user.lang` value from a variable number of records, until the required selectivity was reached. In detail, the TT dataset was edited in three different ways:

- Altering `$.user.lang` to a value other than "en"
- Removing the `$.user.key` values
- Removing only `$.user.lang` key, keeping `user.key`

These edits impact the number of results of the `$.user.lang[?(@ == "en")]` query, as well as some of the internal query execution paths for the libraries. In Figure 6, the results of this analysis are shown. The selectivity of the datasets does not have a significant effect on GpJSON. Node simdjson seems to run slower in the delete cases. This demonstrates that our implementation is still faster than all other implementations and that its performance is independent of selectivity. This is understandable given that GPU-based index construction is executed regardless of selectivity, and, most importantly, the query execution phase of GpJSON is a relatively small part of overall runtime execution.

4.2.2 Varying the file size. Subsequently, we assess how the dataset's size affects the tested implementations' performance. To do this, we alter the size of the TT dataset to be between 0.125× and 16× of the original, resulting in a file size between 100MB and 13GB. Again, we utilize it with the `$.user.lang[?(@ == "en")]` query. Since we set a partition size of 1GB, any dataset bigger than that runs in batch mode.We compare GpJSON running on the three different GPUs and Java JSONPath, Pison, RapidJSON, and simdjson running on the HPC processor. The outcomes of this analysis are shown in Figure 7. GpJSON needs at least 400MB (0.5×) to achieve full performance from the A100, while V100 and P100 need at least 200MB (0.25×) to saturate our implementation. Increasing dataset size does not compromise performance at all in any of the three NVIDIA generations tested. The performance of C++ libraries is not influenced by the different input file sizes. Instead, Java JSONPath performance drops by about 30% for files larger than 2GB (2×).

The above results demonstrate that our implementation outperforms existing low-level, manually tuned, cutting-edge JSON querying frameworks. We demonstrate that the selectivity of a query has no bearing on how quickly we can parse JSON data compared to other state-of-the-art JSON parsers. GpJSON, however, only performs well on relatively large datasets (bigger than 250MB); compared to other libraries, small datasets show a substantial regression. Our implementation shows a performance increase of over 15× over existing JSONPath query libraries available to high-level scripting languages such as JavaScript with a comparable (even less complex) API.

4.3 Experimental Results Vs. RAPIDS

No other libraries exist that can execute JSONPath queries on a GPU. However, JSON parsing and data processing can be achieved on GPUs using NVIDIA's RAPIDS [27] library. RAPIDS does not support running JSONpath queries and can operate only on JSON

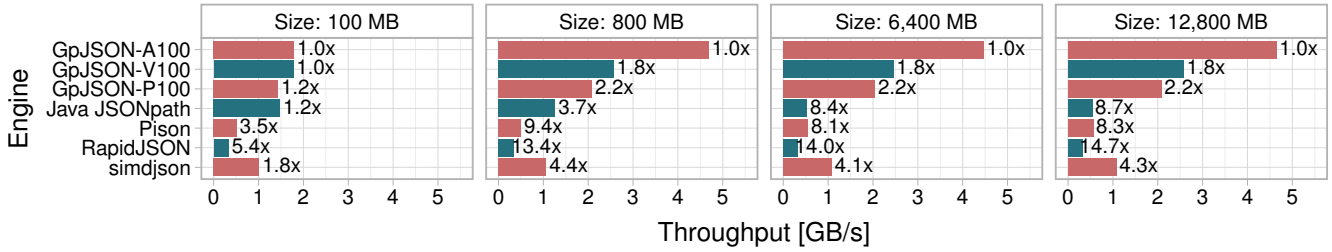

Figure 7: Throughput comparison between GpJSON and other libraries using different dataset sizes. Higher is better.

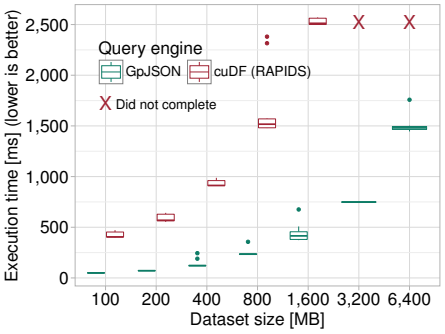

Figure 8: Comparing GpJSON to the cuDF library on the A6000. The cuDF library errored for the two largest files.

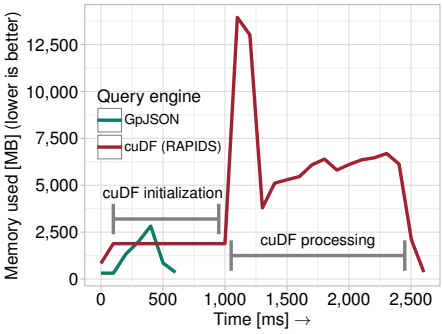

Figure 9: Memory usage of GpJSON vs. cuDF. cuDF execution is further split into two stages.

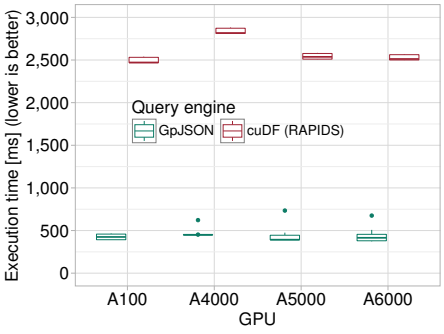

Figure 10: Performance of GpJSON and cuDF across different GPUs. File size was fixed at 1,600MB.

objects with a given JSON schema. In contrast, GpJSON does not require any schema and supports JSONPath natively. Despite the significant differences, it is nevertheless interesting to compare GpJSON against a state-of-the-art GPU-based data processing library. In a new experiment, we analyzed RAPIDS's JSON data processing (using the cuDF library in Python), comparing them against GpJSON. We used the Python interface, as it is the most commonly used and the most mature interface to RAPIDS. cuDF differs from GpJSON in that it is a larger data processing library that is not focused on JSON only. Therefore, it converts JSON data to its own internal in-memory format before processing it. This could result in additional overhead for RAPIDS compared to GpJSON. To more effectively evaluate the performance impact of JSON data conversion in RAPIDS, we conducted specific experiments, as discussed in Section 4.3.2. Overall, GpJSON consistently outperforms RAPIDS in all our experiments.

4.3.1 Varying the file size. First, we investigate the absolute performance difference and the *weak scaling* ability of the two libraries by comparing their performance while querying files of different sizes. We alter the size of the TT dataset to be between 0.125× and 8× of the original, resulting in a file size between 100MB and 6.4GB. We utilize it with the `$.user.lang[?(@ == "en")]` query. Figure 8 depicts the results of this experiment. At file size 100MB, cuDF (405 ms) takes 8x longer to parse the file than GpJSON (50 ms). At file size 1,600 MB, the difference decrease, with cuDF (2,500 ms) taking only 6x longer than GpJSON (415 ms). The difference however is still substantial. Both GpJSON and cuDF consume significant GPU memory. GpJSON streams data in and out of GPU memory as needed to process files while require more than the available GPU memory. cuDF crashes when processing files which require more than the available GPU memory. On an NVIDIA RTX A4000 GPU with 16GB of memory, cuDF fails to parse files larger than 2 GB. When using larger cards such as A5000 (24 GB memory) and A6000 (48 GB), cuDF still encounters memory errors even when free memory is available. Some such errors are already reported [26] and others have not yet been reported. These results demonstrate that GpJSON is more performant and more reliable than the other current available GPU JSON data-processing library. However, cuDF offers more features than just JSON data parsing. Ideally, cuDF should integrate GpJSON and the techniques it proposes into its data parsing infrastructure.

4.3.2 Investigating memory usage. We investigate the memory consumption of cuDF and GpJSON to identify why cuDF performs worse. We record the memory usage of both libraries while they process a 1,600 MB file on a NVIDIA RTX A6000 GPU. We use the nvidia-smi tool to sample memory usage at 6 Hz, the maximum supported by nvidia-smi. Figure 9 depicts the exemplary memory usage of both GpJSON and cuDF from application start to finish. We observe that cuDF takse 2x GpJSON's total execution time just to initialize. cuDF occupies during initialization (1,886 MB) 67% as much as GpJSON takes at its peak (2,824 MB). cuDF has a peak memory usage of 13,948 MB which is 5x GpJSON's peak. The results indicate that cuDF developers should investigate the cause of the

Index name	Size [MB]	Index name	Size [MB]
String index	105.27	Escape index	105.27
String carry index	16.78	Escape carry index	16.78
Newline carry Index	67.11	Leveled bitmap carry	16.78
Newline index	1.24	Leveled bitmap index	210.54

Table 2: Index sizes for Query TT1.

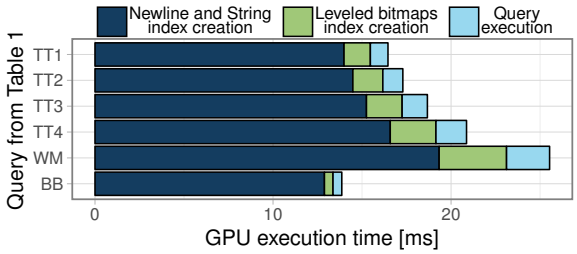

Figure 11: Performance breakdown of queries from Table 1.

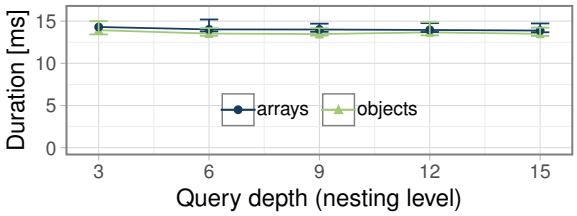

Figure 12: Performance at different query depths.

high initialization time, and the extremely high memory utilization during processing.

4.3.3 Varying the GPU model. We examine whether our results generally hold across multiple GPU models and compare the performance of the two libraries across 4 GPU models. We fix the file size at 1,600MB, close to the largest size that can be processed by cuDF. Figure 10 depicts the results of our experiment. We observe that GpJSON consistently performs 5-6x faster than cuDF on all GPU variants. The results demonstrate that the GpJSON architecture can make different GPUs and their features the same as cuDF, and with higher performance.

4.4 Performance breakdown

In addition to comparing against existing libraries, it is interesting to analyze the performance of GpJSON internals. We first compare the index construction time to the query execution time. Figure 11 shows the performance breakdown for the queries of Table 1 on a 1GB file on the A4000 GPU. As depicted in the figure, index construction, particularly for string indexes, consumes most of the execution time. This result is expected because, as mentioned in Section 3.1, constructing string indexes requires the execution of several GPU intermediate kernels.

The evaluation of the total memory usage of the indexes is another important factor. Table 2 presents the memory consumed by all GpJSON indexes with partition size 1GB on the TT1 query (other queries have similar memory usage). The largest indexes track each input data character, such as string, escape, and leveled bitmap indexes. These indexes are similar in size to SIMD-based solutions that generate structural indexes [30, 31]. In contrast to existing SIMD-based frameworks, however, all GpJSON indexes are exclusively allocated to GPU memory and never accessed from the CPU. This design allows GpJSON to exploit the GPU's very high memory bandwidth during query execution, which is performed by traversing indexes in parallel. Finally, GpJSON handles multiple file lines concurrently using the data partitioning strategy from Section 3.1.1. Consequently, the memory used by indexes of a partition is allocated once and re-used across partitions.

Finally, another interesting aspect is how various query patterns affect GpJSON performance. To evaluate this, we implemented a final experiment using microbenchmarks on JSON objects and arrays at different query depths. The microbenchmark executes queries on two 1GB datasets consisting of uniformly structured nested objects (or arrays), where all JSON objects are identical. We execute queries on both datasets, starting at a single nesting level (i.e., `$.p0` for the first property of an object and `$.[0]` for arrays) and progressively increasing the depth (e.g., `$.p0.p1.p2.p3` for nesting depth 4 in an object, etc.) up to a maximum depth of 15 properties or array elements. Figure 12 presents the results of the microbenchmarks, which illustrate that queries at varying depths generally execute at similar performance, with the query structure having minimal impact. This aligns with Section 6 findings across various selectivity levels, as GpJSON mainly spends its parallel execution time on index construction for all input data, regardless of the query structure, as shown in Figure 11.

5 RELATED WORK

Mison [31] was previously discussed in Section 2.2. It is among the first to leverage structural indexes and SIMD on CPUs for processing JSON data. Simdjson [30] is another very popular SIMD-based JSON parser. It fully parses and validates the document, rather than only extracting some fields without validating them, as Mison does. Unlike Mison, simdjson doesn't allow querying via its API. Users must manually query the JSON tree to extract data, as it lacks support for query languages like JSONPath.Another parallel JSON parsing system, Pison, was proposed by Jiang et al [24]. It is based on methods developed by Mison and simdjson, however, instead of building the structural index sequentially on the CPU, it combines SIMD instructions with multi-threading. The authors claim a speedup of 4.6× over simdjson when all records can be processed in parallel. In our context, however, Pison has a significant limitation. Its parallel implementation does not perform well out of the box for Line-delimited JSON files, and it requires manually handling the parallelism at the line level. In recent work, SIMD-based techniques have been enhanced to handle a wider variety of queries, such as handling descendants [15]. Besides structural indexes, other techniques for JSON parsing exist. Given the inefficiencies of JSON parsing, commercial databases typically store JSON objects using optimized binary formats to increase query performance [32, 33]. A notable exception to this is presented in [11], which introduces several advanced JSON parsing techniques that can be used to query

JSON data directly. The techniques presented in [11] show that it is possible to query JSON data on a CPU with high performance exploiting the properties of the data being parsed. GpJSON, on the contrary, achieves high performance on GPUs without the need to analyze the properties of the data being queried.

Several projects have already proven GPUs' potential in speeding up general-purpose databases. Due to the enormous parallelism made possible by a GPU-based approach, such works demonstrate a significant speedup over conventional CPU-based databases. He et al. [20] show that their GPU-based join algorithms can achieve a performance improvement of 2×-7× over their optimized CPU-based counterparts. Bakkum and Skadron [3] achieve a speedup of at least 20× in SQLite by merely accelerating SELECT SQL queries. Simion et al. [44] demonstrated a speedup of at least 6× for simple queries, where the transfer time from the CPU to the GPU is a significant fraction of the query execution time. Their solution can reach 62×–240× speedup for bigger queries, due to the considerably lower memory transfer overhead. These studies demonstrate that, by converting conventional CPU-based applications to GPU-based techniques, considerable performance improvements can be obtained. This is the reason why several GPU-optimized commercial databases have been proposed over the last few years, such as HeavyDB [21] (previously known as OmniSci), SQreamDB [45], and Kinetica [22]. However, it is worth noticing that these GPU-based database systems usually rely on their own storage or use binary formats, such as Apache Arrow [12] and Apache Parquet [14]. Thus, they cannot process raw files like textual LD-JSON. An interesting solution in this direction is ParPaRaw by Stehle et al. [46]. They use a Deterministic Finite Automaton (DFA) for parsing CSV files. Since each GPU thread processes its chunk in parallel and does not know what the starting point of the DFA should be, data is scanned from all possible starting states. Later, the processed chunk will be combined with the correct parsing state. Another more advanced technique to parse CSV data in parallel is presented in [29]. Unlike with ParPaRaw, [29] parses a CSV file on a GPU by first splitting the input in multiple chunks, and then building parallel prefixes and indexes in parallel. The approach discussed in [29] shares with our approach the idea that input data needs to be split in equally-sized partitions to better leverage the GPU processing power, and the technique to identify new lines in [29] on a GPU is comparable to the one we discuss in Section 3.1.2. Compared to those of [29], the indexes that are built by GpJSON are more complex, as GpJSON needs to construct and index the entire JSON object tree in order to allow queries. In contrast [29] is a pure parser and does not provide any mechanism to query the CSV data, and therefore does not need to build structural indexes. To the best of our knowledge, GpJSON is the first work that leverages structural indexes for parallel parsing and querying of large JSON data using SIMT techniques on GPUs. Moreover, by leveraging the polyglot GraalVM ecosystem, GpJSON is available to multiple high-level dynamic programming languages. The work presented in [25] discusses how to parse JSON objects with a fixed JSON schema [42], which greatly simplifies the task of data de-serialization In contrast, GpJSON can parse any JSON object, regardless of its schema, format, and more in general content. A similar approach is also implemented in the RAPIDS cuDF library [27]. Like with [25], this is a framework that expects the input JSON data to have a well-specified static structure, which is more restrictive compared to what GpJSON supports.

In addition to CPUs and GPUs, specialized hardware has been extensively investigated to accelerate data processing of popular formats (e.g., XML [49, 51]). In recent years, FPGA accelerators have also been explored for JSON. A first approach has been to use FPGAs as co-processors to pre-process data for subsequent CPU usage. Hahn et al. proposed raw filtering [17], which allows to avoid unnecessary parsing by implementing approximate filters for strings, numbers, and structural patterns on an FPGA. Peltenburg et al. [39] developed an FPGA-based technique to convert JSON data (with a known schema) to Arrow. Finally, Dann et al. [8] presented a fully compliant FPGA-based JSON parser inspired by simdjson [30]. These co-processors deliver remarkable performance, but still require CPU involvement for query execution. In contrast, GpJSON's partitioned usage of the GPU memory and its pipelined execution model allow it to perform both parsing and query execution fully on the GPU, entirely eliminating CPU intervention.

Another significant line of research has focused on end-to-end query execution, performing both parsing and querying on FPGAs. In this domain, a notable example has been proposed in [18], and later extended with support for decompression in [19]. In such approaches, the FPGA is specialized for a given JSONPath query, and hardware reconfiguration is needed to execute different queries. Specialized parsing and querying on an FPGA offer excellent performance but sacrifice generality and require careful usage of the FPGA resources. In contrast, GpJSON's bytecode-based query execution engine allows it to execute any arbitrary query on the GPU without the need to re-compile and install a new CUDA kernel for every different query.

6 CONCLUSION

This work introduces GpJSON, a JSON analytics library that can be used to query LD-JSON documents using JSONpath expressions. GpJSON entirely operates on the GPU and relies on the parallel construction of structural indexes. JSONPath query execution in GpJSON is achieved by means of a lightweight query engine operating on the GPU itself, and the partitioning mechanism used by the library makes it possible to pipeline multiple query execution stages, effectively increasing throughput. GpJSON is implemented targeting the GraalVM language runtime, making it available to multiple high-level dynamic programming languages such as JavaScript or Python. In our experiments, we provided an evaluation of GpJSON using datasets obtained from real-world applications, showing that GpJSON outperforms cutting-edge technologies like simdjson, Pison, RapidJSON, Java JSONPath, and RAPIDS in terms of execution time.

ACKNOWLEDGMENTS

We thank Oracle Labs for their support and contribution to this work. The authors from Politecnico di Milano are partially funded by an Oracle research grant. The authors from Vrije Universiteit Amsterdam are funded by the Dutch National Growth Fund 6G FNS and the EU Horizon projects Cloudstars and Graph Massivizer.

REFERENCES

[1] [n.d.]. Java JsonPath implementation. [Online] Accessed: 2024-10-22. Available at: https://github.com/json-path/JsonPath.

[2] [n.d.]. JSONPath Plus. [Online] Accessed: 2024-10-22. Available at: https://github.com/JSONPath-Plus/JSONPath.

[3] Peter Bakkum and Kevin Skadron. 2010. Accelerating SQL Database Operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units* (Pittsburgh, Pennsylvania, USA) *(GPGPU-3)*. Association for Computing Machinery, New York, NY, USA, 94–103. https://doi.org/10.1145/1735688.1735706

[4] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. 2016. Fast Multiplication in Binary Fields on GPUs via Register Cache. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 35, 12 pages. https://doi.org/10.1145/2925426.2926259

[5] Tim Bray. 2014. *The javascript object notation (json) data interchange format*. Technical Report.

[6] Best Buy. 2023. Best Buy Developer API. https://bestbuyapis.github.io/api-documentation/ [Online] Accessed: 2023-03-01.

[7] David Chester. [n.d.]. Robust JSONPath engine for Node.js. [Online] Accessed: 2024-10-22. Available at: https://github.com/dchester/jsonpath.

[8] Jonas Dann, Royden Wagner, Daniel Ritter, Christian Faerber, and Holger Froening. 2022. PipeJSON: Parsing JSON at Line Speed on FPGAs. In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. https://doi.org/10.1145/3533737.3535094

[9] Oracle RDBMS documentation. 2023. JSON Path expressions and simplified syntax. https://livesql.oracle.com/apex/livesql/file/content_I5YTG3IFTTL9XD15PWRG4G4NI.html [Online] Accessed: 2025-06-21.

[10] Gilles Duboscq, Lukas Stadler, Thomas Würthinger, Doug Simon, Christian Wimmer, and Hanspeter Mössenböck. 2013. Graal IR: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*.

[11] Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data*. 445–458. https://doi.org/10.1145/3448016.3452809

[12] Apache Foundation. 2023. The Arrow data format. https://arrow.apache.org/docs/python/json.html Accessed: 2025-06-21.

[13] Apache Foundation. 2023. Loading LD-JSON in Arrow. https://arrow.apache.org/docs/python/json.html Accessed: 2025-06-21.

[14] Apache Software Foundation. [n.d.]. Apache Parquet. [Online] Accessed: 2024-10-22. Available at: https://parquet.apache.org/docs/.

[15] Mateusz Gienieczko, Filip Murlak, and Charles Paperman. 2024. Supporting Descendants in SIMD-Accelerated JSONPath. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 338–361. https://doi.org/10.1145/3623278.3624754

[16] Stefan Gössner. [n.d.]. JSONPath - XPath for JSON. [Online] Accessed: 2024-10-22. Available at: http://goessner.net/articles/JsonPath/.

[17] Tobias Hahn, Andreas Becher, Stefan Wildermann, and Jürgen Teich. 2022. Raw Filtering of JSON Data on FPGAs. In *2022 Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 250–255. https://doi.org/10.23919/DATE54114.2022.9774696

[18] Tobias Hahn, Stefan Wildermann, and Jürgen Teich. 2023. SPEAR-JSON: Selective Parsing of JSON to Enable Accelerated Stream Processing on FPGAs. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. 189–196. https://doi.org/10.1109/FPL60245.2023.00034

[19] Tobias Hahn, Stefan Wildermann, and Jürgen Teich. 2024. JSON-CooP: A JSON Decompression/Parsing Co-Design for FPGAs. In *2024 34th International Conference on Field-Programmable Logic and Applications (FPL)*. 11–18. https://doi.org/10.1109/FPL64840.2024.00012

[20] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[21] HEAVY.AI. [n.d.]. HeavyDB. [Online] Accessed: 2024-10-22. Available at: https://www.heavy.ai/product/heavydb.

[22] Kinetica DB Inc. [n.d.]. Kinetica: The Database for Time & Space. [Online] Accessed: 2024-10-22. Available at: https://www.kinetica.com/.

[23] Twitter Inc. 2023. Developer APIs. https://developer.twitter.com/en Accessed: 2025-06-21.

[24] Lin Jiang, Junqiao Qiu, and Zhijia Zhao. 2020. Scalable structural index construction for JSON analytics. *Proceedings of the VLDB Endowment* 14, 4 (2020), 694.

[25] Krzysztof Kaczmarski, Jakub Narębski, Stanisław Piotrowski, and Piotr Przymus. 2022. Fast JSON parser using metaprogramming on GPU. In *2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)*. 1–10. https://doi.org/10.1109/DSAA54385.2022.10032381

[26] Gregory Kimball. 2019. Error with batched multi-source JSON read with more than one MB per row. https://github.com/rapidsai/cudf/issues/1666. Accessed: 2024-10-22.

[27] Gregory Kimball, Elias Stehle, and Karthikeyan Natarajan. 2023. Reading JSON data in cuDF (NVIDIA). https://developer.nvidia.com/blog/gpu-accelerated-json-data-processing-with-rapids Accessed: 2025-06-21.

[28] Kubernetes. 2023. JSONPath support. https://kubernetes.io/docs/reference/kubectl/jsonpath/ Accessed: 2025-06-21.

[29] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing. In *Proceedings of the 19th Symposium on Database Systems for Business, Technology and Web (BTW 2021)*. 19–38. https://dl.gi.de/bitstreams/d90ece98-9fbe-49af-8778-57595f108478/download

[30] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.

[31] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *Proceedings of the VLDB Endowment* 10, 10 (2017), 1118–1129.

[32] Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon, Hui Chang, Ying Lu, Josh Spiegel, Alfonso Colunga Sosa, Srikrishnan Suresh, Geeta Arora, and Vikas Arora. 2020. Native JSON Datatype Support: Maturing SQL and NoSQL Convergence in Oracle Database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3059–3071. https://doi.org/10.14778/3415478.3415534

[33] MongoDB, Inc. 2019. BSON Binary Data Format Specification. https://bsonspec.org/ Accessed: 2025-06-21.

[34] Rene Mueller and Lukas Stadler. [n.d.]. grCUDA: Polyglot GPU Access in GraalVM. [Online] Accessed: 2024-10-22. Available at: https://github.com/NVIDIA/grcuda.

[35] MySQL. 2023. The world's most popular open source database. https://www.mysql.com/ Accessed: 2025-06-21.

[36] NVIDIA Corporation. 2023. *CUDA Runtime API: Stream Management*. https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html Accessed: 2025-06-21.

[37] Oracle. [n.d.]. GraalVM. [Online] Accessed: 2024-10-22. Available at: https://www.graalvm.org.

[38] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1576–1589.

[39] Johan Peltenburg, Ákos Hadnagy, Matthijs Brobbel, Robert Morrow, and Zaid Al-Ars. 2021. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In *2021 International Conference on Field-Programmable Technology (ICFPT)*. 1–9. https://doi.org/10.1109/ICFPT52863.2021.9609833

[40] Luis Fernando Perez. [n.d.]. Node.js bindings for the simdjson project. [Online] Accessed: 2024-10-22. Available at: https://github.com/JSONPath-Plus/JSONPath.

[41] PostgreSQL. 2023. The World's Most Advanced Open Source Relational Database. https://www.postgresql.org Accessed: 2025-06-21.

[42] JSON Schema. 2023. JSON Schema Standard. https://json-schema.org/ Accessed: 2025-06-21.

[43] Xujie Si, Airu Yin, Xiaocheng Huang, Xiaojie Yuan, Xiaoguang Liu, and Gang Wang. 2011. Parallel Optimization of Queries in XML Dataset Using GPU. In *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*. 190–194. https://doi.org/10.1109/PAAP.2011.30

[44] Bogdan Simion, Suprio Ray, and Angela Demke Brown. 2012. Speeding up Spatial Database Query Execution using GPUs. *Procedia Computer Science* 9 (2012), 1870–1879. https://doi.org/10.1016/j.procs.2012.04.205 Proceedings of the International Conference on Computational Science, ICCS 2012.

[45] SQream. [n.d.]. SqreamDB. [Online] Accessed: 2024-10-22. Available at: https://sqream.com/product/sqreamdb/.

[46] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *Proc. VLDB Endow.* 13, 5 (jan 2020), 616–628. https://doi.org/10.14778/3377369.3377372

[47] Web Technology Surveys. 2023. Usage statistics of LD-JSON for Websites. https://w3techs.com/technologies/details/da-jsonld

[48] Tencent. [n.d.]. RapidJSON Documentation. [Online] Accessed: 2024-10-22. Available at: https://rapidjson.org/.

[49] Jens Teubner, Louis Woods, and Chongling Nie. 2012. Skeleton automata for FPGAs: reconfiguring without reconstructing. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 229–240. https://doi.org/10.1145/2213836.2213863

[50] Walmart. 2023. Walmart Developer Portal. https://developer.walmart.com Accessed: 2025-06-21.

[51] Louis Woods and Gustavo Alonso. 2011. Fast data analytics with FPGAs. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops (ICDEW '11)*. IEEE Computer Society, USA, 296–299. https://doi.org/10.1109/ICDEW.2011.5767669

[52] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 187–204.