HAJPAQUE: Hardware Accelerator for JSON Parsing, Querying and Schema Validation

Samiksha Agarwal
School of Information Technology
Indian Institute of Technology Delhi
New Delhi, India
agarwalsamiksha94@gmail.com

Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

***Abstract*—JSON (JavaScript Object Notation) is quickly becoming the default currency for semi-structured data exchange on the web; hence, it is heavily used in analytics pipelines. State-of-the-art analytics pipelines can now process data at a rate that exceeds 50 Gbps owing to recent advances in RDMA, NVM, and network technology (notably Infiniband). The peak throughput of the best-performing software solutions for parsing, querying, and validating JSON data is 20 Gbps, which is far lower than the current requirement. We propose a novel HW-based accelerator, HAJPAQUE, that ingests 16-bytes of JSON data at a time and processes all the 16 bytes in parallel as opposed to competing approaches that process such data byte by byte. Our novel solution comprises lookup tables, parallel sliding windows, and recursive computation. Together, they ensure that our online pipeline does not encounter any stalls while performing all the operations on JSON data. We ran experiments on several widely used JSON benchmarks/datasets and demonstrated that we can parse and query JSON data at 106 Gbps (@28 nm).**

***Index Terms*—accelerators, ASICs, JSON parsing, query processing, high throughput**

I. Introduction

Over the last few years, JavaScript Object Notation (JSON) has rapidly replaced XML as the preferred format for data representation and exchange on the web, as it is more human-readable and memory-efficient [20]. Major companies such as Twitter and Facebook are now using JSON to transmit web API requests and responses. It has become the default medium of data exchange in high-throughput streaming analytics pipelines such as Azure Stream Analytics by Microsoft [11], Apache Crail, Cloudflare, and Kafka [7], and Amazon Kinesis [2].

Over the years, the processing throughput of these pipelines has increased manifold owing to advances in technology. The current wave is being driven by RDMA over Infiniband, DDR5, and NVMe technologies such as 3D-XPoint. As a result, many of the latest commercially available pipelines support data analytics with a throughput that exceeds 50 Gbps such as Microsoft Azure (@50 Gbps) [11], and Apache Crail(@100 Gbps) [17]. Sadly, we don't have the infrastructure to parse and validate JSON data at these rates. The best software-based systems offer a best-case throughput of roughly 20 Gbps [9], [10].

In this work, we propose HAJPAQUE, a fully-featured hardware accelerator engine for parsing as well as post-processing JSON data; it is capable of sustaining a throughput of about 106 Gbps. In addition to high throughput, HAJPAQUE is a low-latency solution, comprising a 10-stage stall-free hardware pipeline (capable of running at a clock frequency of 833 MHz). Given the plethora of hardware accelerators that sustain such throughputs (albeit for other applications [4], [8], [18]), HAJPAQUE can easily fit in any popular SoC architecture as long as it meets the specifications of advanced analytics pipelines.

We propose novel hardware techniques to tackle the problem of sustaining high rates of parallelism without inducing any stalls in the pipeline. These techniques are **generic in nature** whose potential impact extends well beyond the scope of this paper and can be utilized in hardware designs dealing with any kind of semi-structured data. Unlike SW parsers, the throughput sustained by HAJPAQUE is not susceptible to the nature of JSON data (size of records or depth of nesting/frequency of JSON fields) and the nature of queries (frequency/depth of nesting of queried fields), subject to reasonable limits.

The paper is organized as follows: Section II explains the necessary background, we explain the design of the accelerator in Section III, the evaluation results are shown in Section IV, Section V discusses the related work and we finally conclude in Section VI.

II. Background

A. JSON Language, Query and Schema

A JSON document is a collection of *objects*, where every object is an *unordered* set of *key-value* pairs.

Consider the definition of the JSON format:

```
JSON text = OBJECT...OBJECT
OBJECT = { KEY:VALUE, ... , KEY:VALUE }
KEY = STRING
VALUE = STRING | NUMBER | true | false | null | OBJECT | ARRAY
ARRAY = [VALUE, ..., VALUE]
```

On the lines of Mison [10], we would like to distinguish between the terms *JSON object* and *JSON record*. The term record is used to represent a top-level object. This implies that a JSON record can be a single object or can contain one or more nested objects. In this paper, we assume the uniqueness

of keys within an object (similar to [10]) and do not implement any special mechanism to detect the duplication of keys.

Let us use an example to understand the JSON language.

```
{"Name": "John", "Age": 50, "Car": NULL}
{"Name": "Ray", "Age": 45, "Car":{"attributes":{"color": "Black",
"electric": Yes}}}
```

This example consists of two JSON records. Each record has three top-level key-value pairs. The first record does not contain any nested JSON object. The second record represents an example of multi-level JSON object nesting.

Currently, an adaptation of the XML query language XPath in the context of JSON (called JSONPath [3]) is quite popular and is predominantly used for querying JSON data. In this paper, we will refer to a JSONPath query as a *JSON query*.

A JSON Query consists of a sequence of JSON keys, each of which is referred to as a *query key*. Let us understand JSON queries using two distinct queries for the same example.

```
Query1: "Name"
Query2: "attributes.electric"
```

The first query consists of a single query key ("Name") and its results are expected to be "John" and "Ray". The second query consists of two query keys ("attributes" and "electric"). The queried keys appear only in one of the two JSON records and the result of this query is "Yes".

There are popular draft proposals for a formal schema: JSON Schema [3] and the grammar by Pezoa et al. [12]. The schema rules specified below are examples of some of the JSON Schema rules.

```
Rule1: The value of the "Name" key must be a string
Rule2: The value of the "Age" key must be a number and must lie in the
range 10-100
Rule3: The following keys are mandatory in every JSON record: "Name"
and "Age"
```

III. Design Implementation

A. Overview

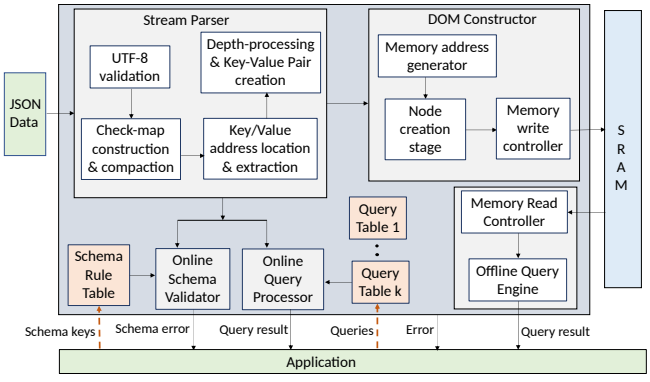

Fig. 1. Design of HAJPAQUE

Figure 1 shows the block diagram of the proposed accelerator engine, which includes the stream parser, parse tree (Document Object Model (DOM)) constructor, query processors (online and offline) and the online schema validator. Along with traditional offline querying, we also support high-throughput online query processing; they are integral features of modern analytics pipelines [5], [6], [19].

B. Parsing engine

1) UTF-8 Character Checking: JSON data is received by HAJPAQUE in blocks of 16-bytes at a time, which we refer to as a *set*. Each byte is encoded as either a leading byte or a continuation byte and the character-length information is associated with every leading byte. All the bytes of a set are simultaneously analyzed to check whether they conform to the UTF-8 encoding standard. In case of an error, the application is informed using an UTF-8 encoding error signal and the bytes are ignored.

2) Locating and Extracting the Fields: This submodule finds the presence of a key or value or both in a set of 16 bytes and subsequently finds their exact location. To start with, we create a new structural token, called a *qcolon*. Whenever a quotation mark is immediately followed by a colon (we assume there are no whitespaces between quote and colon), these two structural tokens are unified and referred to as a single *qcolon (":)*. If a quotation appears without a succeeding ':' we refer to it as a *quote*. The aim is to find the locations of all the *quotes* and *qcolons* in the set *in parallel*.

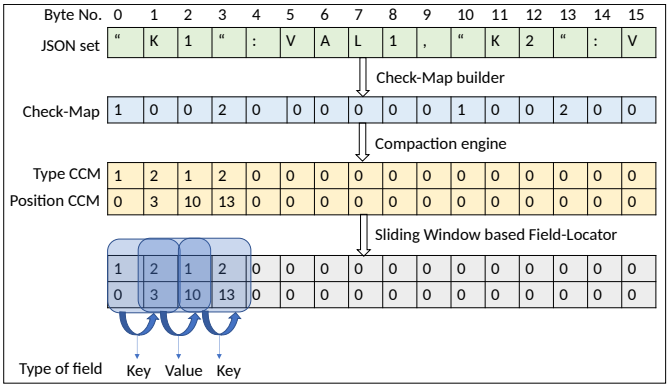

Fig. 2. Stages involved in locating JSON fields

To proceed, let us define a *check-map*, which is a 16-bit vector of 2-bit values. A 0 represents the fact that at a given location we do not have a token of interest. 1 represents the presence of a *quote* and 2 represents the presence of a *qcolon* at the corresponding byte location in the set of 16 bytes. Let us now define a *compacted check-map (CCM),* which is a vector of 2-bit values. The i^{th} entry indicates the position and type of the i^{th} structural token of interest: *quote* or *qcolon*. The check-map can be generated in parallel by 16 parallel logic blocks. Refer to Figure 2 for the rest of the discussion.

❶ The compaction engine takes in the check-map as input and creates the CCM, which consists of two compacted sub-checkmaps: a *type check-map* to indicate the type of the token of interest (*quote* or *qcolon*) and a *position check-map* to indicate the actual position of the token in the set.

Subsequently, we create 4 logic blocks – each one of them considers 4 positions in the check-map. For example, the first block considers positions $\langle 0, 1, 2, 3 \rangle$, the second considers positions $\langle 4, 5, 6, 7 \rangle$, and so on. Each logic block reads the 8 bits assigned to it (2 per position), and accesses a lookup table that stores the corresponding CCM. Now, we have four CCMs. We then proceed to create a tree-like structure, where each internal node takes in the CCMs of its two children and *merges* them to create a combined CCM using a lookup table. The final output is the CCM for the entire set of 16 bytes.

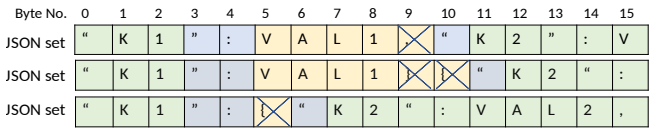

Fig. 3. Filtering the value fields

❷ This CCM is then given to the *Field Locator* engine, which examines each pair of consecutive positions of the check-map using a 2-entry sliding window. Note that this sliding window is applied to the compacted check map in a parallel manner instead of sequential manner. This means that we have one logic block for analyzing positions 0 and 1, one more for positions 1 and 2, and so on. Using this parallel sliding-window based analysis, the CCM is decoded to determine the location of the keys or values or both, if present in a set. We then use an elaborate set of rules to derive the meaning of token pairs. There are many possible combinations. We are listing a few simple ones in the table shown below. A qcolon followed by another qcolon is not valid JSON syntax, hence a syntax error is flagged to the application and the field enclosed between the two qcolons is excluded from further processing. The *value* might contain a few structural tokens like a comma or curly braces; they need to be filtered out later (see Figure 3).

⟨**current token, next token** ⟩	**type of the field**
quote, *qcolon*	key
quote, *quote*	value
qcolon, *quote*	value
qcolon, *qcolon*	error

The information of the presence of a key or value using the last valid structural tokens of the CCM is registered and then used while analyzing the first structural token of the next set's CCM; this is because a field may straddle sets.

❸ This sub-module works on the *start* and *end* addresses of the keys/values received from the previous stage and extracts the corresponding field. A single field often does not start and end in the same set, resulting in the need to buffer bytes and store the incomplete key/value of the previous set until the ending location of the key/value is found, so that the entire field can be extracted as a whole. In case, the length of the key or the value to be buffered exceeds the size of the buffer, a buffer overflow error is indicated to the application and the key/value that caused the buffer to overflow is excluded from further processing in entirety. Each key is stored in an array of keys, where the i^{th} entry stores the i^{th} key in the set. Likewise is the case for values. The underlying hardware structure essentially implements a 2D array of characters. These arrays are not *compacted*.

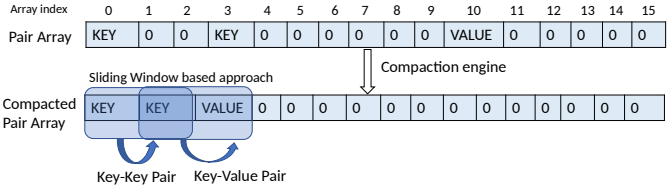

Fig. 4. Establishing relationships between JSON fields

3) Depth Processing and Key-Value Relationship Building: Once the keys and values are extracted, the next task is to establish the relationships among different fields namely the parent-child relationships among nested keys and find key-value pairs. A dedicated depth-processing engine calculates the depth of every extracted field with the help of the CCM of the left and right curly braces. A global register stores the current depth information for a set, which is then passed to the next set and serves as the starting depth value for the next set.

Once the depth information is available to us, we need to know the actual order of the extracted keys and values in the raw JSON data stream in order to establish the parent-child relationships between the fields. Hence, we *merge* the arrays corresponding to keys and values to form a new array called the *Pair Array*. Once again, we use the compaction engine to compact the Pair Array. Then we use a 2-entry sliding-window in parallel to examine consecutive entries of the Pair Array and build parent-child relationships between consecutive keys and form key-value pairs(see Figure 4).

⟨**current field, next field** ⟩	⟨**current depth, next depth** ⟩	**Type of current field**
key, *key*	*d*, *d+1*	Non-terminal key
key, *value*	*d*, *d*	Terminal key

If the i^{th} position of the Pair array has a key and the $i+1^{th}$ position has a value, then the key at the i^{th} position becomes a *terminal* key, if the depth of the key and the value are same. We can say that the key and the value at the i^{th} and $i+1^{th}$ positions, respectively, form a *Key-Value Pair*. If there is a key at the i^{th} position followed by another key at the $i+1^{th}$ position of the Pair Array, then the key at the i^{th} position becomes a *non-terminal* key, if its depth is one less than the depth of the succeeding key.

The **novel contribution** of this section is the parallel application of sliding-windows, which is used extensively, and is responsible for the achieved parallelism. The outputs of the stream-parsing engine can be utilized by either the online post-processing blocks like the online query analyzer or schema validator; they can also be passed to the DOM parser. The DOM parser constructs a parse tree in memory, which can then be accessed by any application later. An offline query engine has also been implemented, which can traverse the tree stored in memory and return the results of an application

query. We omit a detailed discussion on the implementation of the conventional DOM parser due to a lack of space.

C. Online Query Processor

The online query processing engine receives parsed keys and values from the Stream Parser and then evaluates a set of queries (standard single-variable queries [3]). The assumption is that before the JSON data begins to stream into HAJPAQUE, the queries from the application are compiled and loaded into the hardware accelerator. We use a register-based data structure called the *Query Table* for each query (see Figure 5). For each entry in the Query Table, the *Valid* bit indicates whether the corresponding entry of the table contains a valid query key – note that a query can contain a sequence of keys, where a key is the child of the previous one in the list. The sequence of keys need not start from the root of the entire JSON document. The query key with the lowest depth (say d) fills the first entry of the Query Table, the key with depth $d+1$ fills the second entry of the Query Table, etc.

In case the Query Table is re-programmed at runtime, the new queries will only be applicable to the JSON stream data that arrives at the accelerator post re-programming.

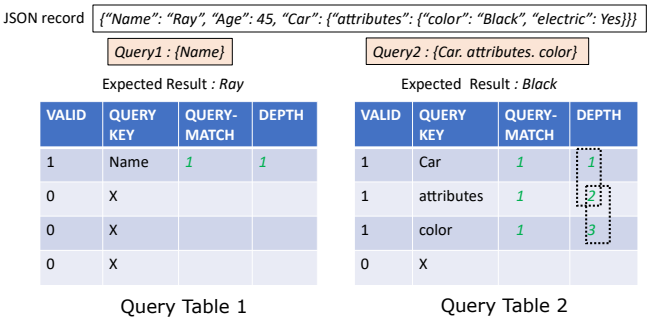

Fig. 5. Example a of Query Table

We propose a **novel hardware unit** called the *Highly-Parallel Query Matching Engine (HPQME)* for query processing. Every key in a JSON set (terminal or non-terminal) is matched with all the entries of the Query Table in parallel. Note that every query has to end at a terminal key.

In the case of a successful match, the *Query-Match* bit is set to 1 for the corresponding entry of the Query Table, provided the entry is valid. Before setting the *Query-Match* bit of a Query Table entry, the following conditions are also checked: ❶ If the match is due to a non-terminal key, the *Valid* bit of the succeeding entry of the Query Table must be '1' ❷ If the match is due to a terminal key, the *Valid* bit of the succeeding entry in the Query Table must be '0', except if the match is for the last table entry. These checks are important to ensure the correctness of the pre-compiled query. In case, any of these conditions are not satisfied, a query processing error is signaled to the application and the *Query-Match* bits of all the entries of the Query Table are reset.

When the *Query-Match* bits corresponding to all the valid entries of the Query Table are set to '1', a valid result of the query is generated. The result of the query is the "value" field of the terminal key that caused the *Query-Match* bit of the last valid entry of the Query Table to be set to 1.

Despite a successful match, the result can be erroneous in case the order of keys in the data is not the same as the order desired in the query. To overcome this challenge, we make use of the depth of the parsed keys calculated by the Depth Processor. Whenever there is a match between the incoming key and the key stored in the Query Table, the depth of the key is also stored in the Query Table in the corresponding query entry. When the *Query-Match* bits of all the valid table entries are set to '1', we examine the stored depth information of the keys. We apply a 2-entry sliding-window in parallel for each pair of consecutive entries of the Query Table. A valid response to the query is generated by the Query Processor if and only if the following condition is satisfied:

If the Query Table has k valid entries,
Depth[i] = (Depth[i-1] + 1) where (i=0,1,...k-1)

D. Schema Validator

In this work, we implement the three schema rules (shown as an example in Section II). For checking Rules (1) and (2), the hardware matches each of the terminal keys with the *reference key*. In the case of a successful match, it examines the value field of the matching key. All the characters of the value field are checked to find if it is a string or a number, and in case any character is found to be not in the allowed set of characters, a schema validation error is flagged.

For checking Rule (3), it is important to identify the starting point of a new JSON record because the checking of mandatory keys must happen for each record. We create a set of parallel, 2-entry sliding windows that consider consecutive pairs of entries in the CCM and look for the '}{' pattern. For every new record, all the parsed keys of a set are verified against the mandatory reference keys using a *Schema Match* table (see Figure 6).

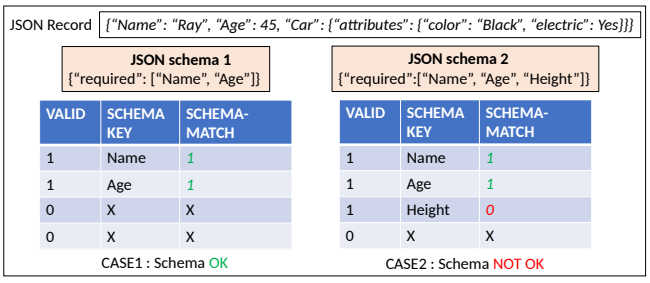

Fig. 6. Examples of Schema Validation

IV. Results and Performance Evaluation

A. Experimental Setup

The design was coded in Verilog, functionally verified using the Xilinx Vivado simulator and synthesized using the Synopsys Design Compiler (DC) using 28nm (ASIC) technology libraries of a major silicon vendor. All the results

have been reported using the technology library for the slow-slow(SS), low-voltage(0.75 V) and low temperature(-40 °C) corner (worst case scenario for this technology). All the software was run on a desktop-based system with an i7-8700 CPU running at 3.20GHz with the Ubuntu Linux 18.04.5 LTS OS. We assume an SRAM of size 1 MB for storing the DOM tree representation of the JSON datasets (larger trees can overflow into lower-level memory).

B. Datasets

We tested our design using standard datasets, which have been extensively used in prior work [9], [10]. Table I shows the different attributes of these datasets. The throughput values *saturate* for data sets greater than 10 KB; hence, there is no point in considering larger data sets.

TABLE I
CHARACTERIZATION OF JSON DATASETS. VALUES NORMALIZED TO 10 KB CHUNKS.

JSON dataset	**Number of fields (keys / values) to be parsed**	**Maximum depth of nested fields**	**Number of JSON records**
iris.json	976	1	99
covid_features.json	600	2	33
business.json	870	3	13
twitter.json	410	6	2

C. Synthesis Results

We found that the design operates correctly without any timing violations (at the slowest process corner) for clock cycle times of 1.2 ns and more. Hence, for reporting all the results in this paper, we operate our design at the maximum clock frequency of 833 MHz (cycle time=1.2 ns).

TABLE II
DESIGN PARAMETERS USED FOR REPORTING RESULTS

Parameter	**Value**
Number of Query Processors/Tables	1
Number of entries in the Query Table	7
Width of a Query Table entry	32 bytes
Number of entries in the Schema Table	4
Width of a Schema Table entry	32 bytes

For reporting the area/power figures, we configured our design as per the parameters mentioned in Table II. It is assumed that the designer has an estimate of the sizes of structures required. Just in case, more entries are required, we can handle these cases in software. Given that such tables are not in the critical path, the clock frequency remains the same even when we scale the number of Query or Schema Table entries to 64. With these sizes we were able to process all our data sets.

Table III summarizes the area and power consumption figures of the design. We observe that HAJPAQUE is a combinational-logic intensive design, as it is highly parallel in nature and extensively utilizes combinational-logic resources to achieve this parallelism. The sequential logic area required for HAJPAQUE is primarily attributed to the pipeline registers (10-stage pipeline).

TABLE III
AREA AND POWER CONSUMPTION OF HAJPAQUE

Total Area	292839 μm^2
Combinational Area	184097 μm^2
Non-combinational Area	108742 μm^2
Static Power	0.008 mW
Dynamic Power	43.31 mW

D. Performance Analysis

1) Parsing Throughput: The first experiment evaluates the throughput for varying sizes of JSON datasets. We show the results for *twitter.json*, whose size was varied from 100 B to 0.6 MB.

Table IV reports the throughput provided by HAJPAQUE for DOM-based parsing and stream parsing, respectively. As the datasets become larger (>10 kB), the throughput values saturate at 98.4 Gbps and 106 Gbps, respectively (at an operating frequency of 833 MHz). This happens as fixed overheads such as the pipeline latency become insignificant as compared to the total execution time. It is fair to consider large files of size >10 kB for evaluating the throughput since practical JSON applications do not deal with very small datasets. Moreover, **there is no need** to evaluate our design with datasets larger than the ones we have used; we achieve steady state performance with our datasets. In the case of DOM-based parsing,the achieved CPB (cycles per byte) is slightly lower than the theoretical CPB of 0.0625 (1/16) cycles per byte due to the memory accesses for storing the parse-tree representation of the JSON data in memory.

TABLE IV
CPB/THROUGHPUT FOR DIFFERENT SIZES OF DATASETS (CPB IS CLOCK CYCLES PER BYTE). WE APPROACH A STEADY STATE AFTER 10 KB.

Dataset size	DOM parsing		Stream parsing	
	CPB	**Parsing throughput**	**CPB**	**Parsing throughput**
100 B	0.18	37.0 Gbps	0.16	41.6 Gbps
1 kB	0.079	84.4 Gbps	0.072	92.6 Gbps
10 kB	0.0685	97.2 Gbps	0.0634	105.1 Gbps
100 kB	0.0679	98.1 Gbps	0.0625	106.6 Gbps
0.6 MB	0.0677	98.4 Gbps	0.0625	106.6 Gbps

A key feature of the stream parsing engine of HAJPAQUE is that its performance is not impacted by the nature of the JSON data or the JSONPath queries. We examined the impact of the nature of JSON datasets on the performance of the stream parser using 10 KB of JSON data from each of the four JSON datasets (shown in Table I). We also studied the impact of the variation in the nature of queries on the performance of HAJPAQUE (we show the results for 10 kB of the *business.json* dataset by evaluating three different types of queries on it). The queries vary from each other in terms of the frequency and depth of the valid results (see Table V). For both the cases, we do not see any impact on the

CPB/throughput of the stream parser. This is expected because there are no pipeline stalls due to memory accesses in the stream parser. Hence, even if there are more JSON fields to be parsed/queried, we will still not have any stalls. Further, the depth of the parsed/queried fields is calculated by the depth processing module, whose performance is not impacted by the maximum depth of the field of the JSON keys. Of course, there are limitations set by the number of entries, buffer sizes, etc. However, these values can be increased at design time with a minimal impact on the cycle time.

TABLE V
VARIATION OF CPB WITH QUERIES IN THE STREAMING MODE

***business.json* query**	**Number of valid query results**	**Maximum depth of the queried field**	**CPB**
business_id	13	1	0.0625
attributes.ambience.touristy	3	3	0.0625
hours.Sunday	6	2	0.0625

2) Comparison with the State-of-the-art: Next, we compare HAJPAQUE with other state-of-the-art JSON parsers.

TABLE VI
COMPARISON OF SUPPORTED FEATURES AMONG SEVERAL PARSERS

Feature	**RapidJSON**	**Mison**	**SimdJSON**	**HAJPAQUE**
Stream Parsing	✓	✓	✓	✓
Online Querying	✓	✓	✓	✓
DOM parsing	✓	✗	✓	✓
Offline Querying	✓	✗	✓	✓
UTF8 validation	✓	✗	✓	✓
Schema Validation	✓	✗	✗	✓

Let us now compare the throughput of different types of parsing in Table VII and compare the results with two of the fastest software parsers: SimdJSON [9] and RapidJSON [15].

TABLE VII
THROUGHPUTS FOR PERFORMANCE BENCHMARKING (DATASET *twitter.json* AND THE QUERY "USER.ID")

Parser	**Parse (create DOM only)**	**Parse+Scan DOM**	**Stream-Parse + Process query online**
RapidJSON [15]	1.7 Gbps	0.8 Gbps	1.6 Gbps
SimdJSON [9]	5.0 Gbps	2.6 Gbps	4.8 Gbps
HAJPAQUE	98.4 Gbps	60 Gbps	106.6 Gbps

The absolute throughput of HAJPAQUE reduces from 98.4 Gbps to 60 Gbps since the time taken in scanning the parse tree is dependent on the number of memory accesses, which is proportional to the number of nodes in the tree-based representation of the JSON document. In SimdJSON, the time spent in the UTF-8 validation stage depends on the fraction of non-ASCII characters in the input data, and the time spent in locating key/value pairs depends on their frequency. However, in the case of HAJPAQUE, the UTF-8 validation stage and the key/value location stage take the same number of cycles, irrespective of the nature of the input data. If we do not consider DOM-tree construction, then the rest of our online pipeline does not have any stalls.

V. RELATED WORK

Traditional JSON parsers such as GSON [13], JACKSON [14] and RapidJSON [15] use state-machine based algorithms, which execute a series of instructions on input data, byte by byte. Recent research has focused on the use of modern processors to leverage SIMD-level parallelism to quickly locate queried fields without having to perform expensive lexical analysis such as Mison [10] and SimdJSON [9]. They can sustain a peak throughput in the range of 16-24 Gbps.

HPXA [1] proposes a parallel approach for XML parsing that ingests data 16 bytes at a time, and demonstrates a parsing throughput of roughly 100 Gbps on a 28 nm technology node (for ASICs). However, its role is just limited to DOM parsing without state-of-the-art stream parsing and online post processing; it does not have the elaborate structures that we have such as the sliding windows, lookup tables, and deep-parallel searches for parsing, querying and schema validation. An FPGA-based implementation of a streaming JSON parser has been proposed in 2017 [16], which makes use of a deterministic automaton for parsing and another automaton for evaluating a set of JSONPath predicates. The message throughput of the proposed system along with the JSON parser has been shown to be to about 4 Gbps.

VI. CONCLUSION

We were able to create a fully-featured JSON parser by using three hardware-based innovations: parallel sliding window-based field relationship builder, parallel query processing engine that uses depth information, and a method for finding the beginning and end of records by processing the CCM in parallel. They ensure that we do not have to introduce any stalls; hence, we can achieve the maximum theoretical throughput for sufficiently large datasets ($>$ 10 KB). The methods introduced in this paper have a generic scope and can be directly used to accelerate XML/HTML (and similar semi-structured data) parsing, schema validation and query processing.

REFERENCES

[1] I. Ahmad, S. Patil, and S. R. Sarangi, "Hpxa: a highly parallel xml parser," in *DATE*, 2018.
[2] AWS. (2013) Amazon kinesis. https://aws.amazon.com/kinesis/.
[3] P. Bourhis, J. L. Reutter, F. Suárez, and D. Vrgoč, "Json: data model, query languages and schema specification," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems*, 2017, pp. 123–135.
[4] Y. Fang, C. Zou, A. J. Elmore, and A. A. Chien, "Udp: a programmable accelerator for extract-transform-load workloads and more," in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2017, pp. 55–68.
[5] A. S. Foundation. Apache storm. http://storm.apache.org/.
[6] A. S. Foundation. Spark streaming. https://spark.apache.org/streaming/.
[7] A. S. Foundation. (2017) Apache kafka. https://kafka.apache.org/.
[8] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "Hare: Hardware accelerator for regular expressions," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[9] G. Langdale and D. Lemire, "Parsing gigabytes of json per second," *The VLDB Journal*, vol. 28, no. 6, pp. 941–960, 2019.
[10] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann, "Mison: a fast json parser for data analytics," *Proceedings of the VLDB Endowment*, vol. 10, no. 10, pp. 1118–1129, 2017.
[11] Microsoft. Azure documentation. https://docs.microsoft.com/en-us/azure/data-factory/copy-activity-performance.
[12] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte, and D. Vrgoč, "Foundations of json schema," in *WWW*, 2016.
[13] G. repository. Google gson. https://github.com/google/gson.
[14] G. repository. Jackson. https://github.com/FasterXML/jackson.
[15] G. repository. rapidjson. https://github.com/miloyip/nativejson-benchmark.
[16] D. Ritter, J. Dann, N. May, and S. Rinderle-Ma, "Hardware accelerated application integration processing: Industry paper," in *ICDEBS*, 2017.
[17] P. Stuedi, A. Trivedi, J. Pfefferle, R. Stoica, B. Metzler, N. Ioannou, and I. Koltsidas, "Crail: A high-performance i/o architecture for distributed data processing." *IEEE Data Eng. Bull.*, vol. 40, no. 1, pp. 38–49, 2017.
[18] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, "Hawk: Hardware support for unstructured log processing," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 2016, pp. 469–480.
[19] WSO2. Wso2 stream processor. https://wso2.com/integration/streaming-integrator/.
[20] S. Zunke and V. D'Souza, "Json vs xml: A comparative performance analysis of data exchange formats," *IJCSN International Journal of Computer Science and Networks*, vol. 3, no. 4, pp. 257–261, 2014.