dsJSON: A Distributed SQL JSON Processor

MAJID SAEEDAN, University of California, Riverside, USA
AHMED ELDAWY, University of California, Riverside, USA
ZHIJIA ZHAO, University of California, Riverside, USA

The popularity of JSON as a data interchange format resulted in big amounts of datasets available for processing. Users would like to analyze this data using SQL queries but existing distributed systems limit their users to only two specific formats, JSONLine and GeoJSON. The complexity of JSON schema makes it challenging to parse arbitrary files in a modern distributed system while producing records with unified schema that can be processed with SQL. To address these challenges, this paper introduces dsJSON, a state-of-the-art distributed JSON processor that overcomes limitations in existing systems and scales to big and complex data. dsJSON introduces the projection tree, a novel data structure that applies selective parsing of nested attributes to produce records that are ready for SQL processors. The key objective of the projection tree is to parse a big JSON file in parallel to produce records with a unified schema that can be processed with SQL. dsJSON is integrated into SparkSQL which enables users to run arbitrary SQL queries on complex JSON files. It also pushes projection and filter down into the parser for full integration between the parser and the processor. Experiments on up-to two terabytes of real data show that dsJSON performs several times faster than existing systems. It can also efficiently parse extremely large files not supported by existing distributed parsers.

CCS Concepts: • **Computing methodologies** → **MapReduce algorithms**; • **Information systems** → **Relational database model**; **Semi-structured data**.

Additional Key Words and Phrases: JSON, Spark, SQL, JSONPath, schema inference, distributed processing

ACM Reference Format:
Majid Saeedan, Ahmed Eldawy, and Zhijia Zhao. 2023. dsJSON: A Distributed SQL JSON Processor. *Proc. ACM Manag. Data* 1, 1, Article 103 (May 2023), 25 pages. https://doi.org/10.1145/3588957

1 INTRODUCTION

The JavaScript Object Notation (JSON) is an open data-interchange format. It gained its popularity with the rise of web applications, since it is the native representation of data in JavaScript. As a result, it has been adopted in various applications resulting in the need for storing and processing very big JSON data.

Consider the JSON file snippet in Figure 1. A user is interested in extracting the objects (enclosed by `{` and `}`) in the `products` array, that fall within a specific category (`category == 11`). To avoid expensive serial parsing, which can take hours for terabytes of data, the file should be partitioned and parsed in parallel by different machines, as shown in the figure. However, this is extremely challenging because, for the second and subsequent partitions, the parsing state needs to be determined in the middle of the file. Moreover, as the attributes (key-value pairs) of an object might be split among two (or more) partitions, it is non-trivial for the parser to examine the filtering conditions specified by the user. Even worse, since big data frameworks, like Spark [3]

Authors' addresses: Majid Saeedan, msaee007@ucr.edu, University of California, Riverside, Riverside, California, USA, 92521; Ahmed Eldawy, eldawy@ucr.edu, University of California, Riverside, Riverside, California, USA, 92521; Zhijia Zhao, zhijia@cs.ucr.edu, University of California, Riverside, Riverside, California, USA, 92521.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2023 Copyright held by the owner/author(s).
2836-6573/2023/5-ART103
https://doi.org/10.1145/3588957

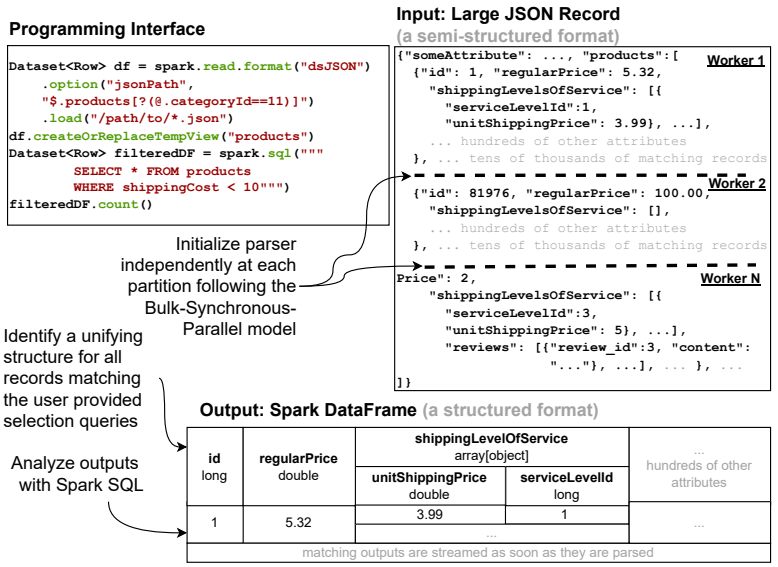


Fig. 1. An example highlighting dsJSON processing steps

and AsterixDB [9], use the bulk-synchronous-parallel (BSP) model, the parallel parsers are not allowed to communicate asynchronously while parsing. To be able to perform different types of analysis readily available like SparkSQL queries, the JSON processor should build a unified schema that matches the user provided queries. In this case, it builds a schema for all products that fall within the specified category. Then, various SQL queries can be easily applied on the extracted records. The example on the top-left highlights a simple query that counts the extracted records with `shippingCost < 10`.

JSON data is produced and consumed in a large variety of use cases. As a popular data interchange format, it is widely used to store data logs, backups, in raw text format, usually encoded in UTF-8 or other suitable text encodings. It is also usually used to store big data dumps that are collected from web APIs, or scraping websites. Furthermore, JSON is used to represent datasets generated by professionals from various fields, like GeoJSON which is a popular format for Geospatial datasets, thanks to its flexibility in metadata storage.

Due to the vast range of use cases, JSON data often contains complex nested structures and can be very large in size, posing several challenges. Unfortunately, existing tools have limitations that can make it difficult to process JSON data in certain situations, leaving users with the option of creating their own ad-hoc programs to extract the necessary data. However, this process can be time-consuming, and even the final program may still have issues due to the complexity of the problem. The following section outlines the three primary limitations of current state-of-the-art JSON processors.

L1. Limited flexibility for parsing complex nested JSON records. Existing tools do not provide full flexibility to the user to determine the most relevant parts that should be parsed and included in the output. Without this flexibility, processing big JSON data can be impractical, due to memory and time constraints. The simplest JSON parsers simply parse the entire input to one large object in memory. Others provide a little more flexibility by taking a simple query from the user and extract values that match it. However, it is still not possible to select multiple attributes from different nesting levels. Hence, the user is still limited in defining the values to be included in the output.

Distributed systems like Apache Spark don't support all use cases due to lower flexibility at the parsing stage.

L2. Limited scalability for data in the general JSON format. Existing distributed systems like [3, 9, 21] cannot parse the general JSON format, instead they are limited to a specific type like the JSONLines [30] format, in which records are separated by the new line character, where each line corresponds to one record in the output. However, many real applications require processing general JSON files like the one shown in Figure 1, which can grow in size, making it a huge bottleneck when processed in a single machine environment. Existing parallel parsers of the general JSON format are designed for parallel shared-memory systems or asynchronous message passing interface (MPI) and cannot be used in big-data shared-nothing systems such as Apache Spark and AsterixDB which rely on the BSP model where asynchronous communication is not allowed. This makes it impossible to process very large files where the resources of a single machine are not sufficient.

L3. Limited support for reading JSON data to a structured format to support executing analytical queries. Performing analysis, like executing SQL queries or training a machine-learning model, requires identifying a unified structure of the data. This can be challenging on its own. Systems like Spark perform an inference step to identify this structure by simply computing the union of the object type of each JSONLine in the input data. However, this would fail when there are hundreds of thousands of different attributes in all records which is too big to handle for Spark as further detailed in Section 6. Furthermore, existing single-machine parsers that support the general JSON format are not integrated with a full-featured data analytics system, making it very difficult to perform complex analysis without applying intermediate steps like converting the data into a different format.

Simply converting data to the JSONLines format is not always practical. First, there is no unique way to do this conversion. In the example in Figure 1, a user interested in product reviews will only extract the reviews which include the review score and text, while a user interested in price analysis will extract product data including category, price, and description. Second, there is no scalable way to convert general JSON to JSONLine as mentioned in L2. Ensuring data is generated as JSONLines is not always possible when data is collected from various sources. Third, even if data is strictly collected in this format, existing JSONLine processors could fail with complex records since they provide very limited options for schema inference, like identifying a schema for a specific product category, or simplifying complex schema.

Moreover, it is worth noting that the data we are concerned with is stored in raw text format, encoded using UTF-8 or similar, making it even more challenging to process, since the start and end of each record and the number of attributes it contains is not known prior to parsing. This is in contrast to data stored in managed databases like [2, 38] that can efficiently process JSON records by storing them in a binary format and possibly building indexes around them, after a required data insertion step.

To address the above limitations and challenges, this work introduces the first Distributed SQL JSON (dsJSON) processor, fully integrated into an analytics system (Apache Spark). It parses any valid JSON file into a set of records (Dataframe) that can then be processed using SQL queries or machine learning models. In mitigating L1, dsJSON proposes a data structure called the projection tree, that is used to track the parsing state, and selectively extract the desired attributes, which ensures minimal overhead. The projection tree is built from user provided queries, providing a lot of flexibility for the user to define the parsing stage. To overcome L2, dsJSON provides two partitioning techniques that make it possible to initialize and process the parser at each partition fully independently, following the BSP model. The minimal memory requirement, in addition to the robust partitioning methods, make dsJSON easily scale to very big JSON data. To address L3, dsJSON provides a schema inference method using the proposed projection tree. Building a

schema, based on the selectively parsed records, makes it possible to handle very complex records. Schema inference is required for integration with systems that expect (semi-)structured data, like SparkSQL [10] and MLlib [35]. Furthermore, dsJSON pushes SQL projection and filter operations into the parser to improve the performance by skipping unnecessary parts of the input file.

The rest of this paper is organized as follows. Section 2 provides some preliminaries about JSON grammar and parsing. Section 3 introduces the projection tree and provides an overview of the system. Section 4 describes how we build the initial projection tree for the user provided JSONPath queries. Partitioning and schema inference are discussed in Sections 5 and 6, respectively. Section 7 discusses SQL projection and filter push down into the projection tree. Section 8 gives the details of record parsing. Section 9 provides an extensive experimental evaluation of the system. Section 10 describes the related work. Finally, Section 11 concludes the paper.

2 PRELIMINARIES

The standard JSON grammar has two main components: objects and arrays. An object can have multiple key-value pairs. The order of the pairs has no significance, and this can add to the complexity of parsing the records into a structured, uniform format. The other major component is arrays. The elements of arrays can be any valid JSON value. A value in JSON can be any nested JSON record, or a primitive value. Primitives can be strings, numbers, or one of the reserved words: `true`, `false`, and `null`. The latest version of the standard JSON grammar can be accessed at [29].

There are many variants of the JSON syntax. One very popular variant is the JSONLine format, in which each line in the file contains a separate JSON object. While this format imposes more restrictions over the general JSON format, it is adopted in some systems, e.g., Apache Spark [3] and AsterixDB [9], due to the simplicity of parsing it in parallel. Simply, the file is partitioned into lines and each line is parsed as one unit. Another well-known variant of the JSON format is GeoJSON. This format follows the conventions of the standard JSON format, and the records can exist in multiple lines. It follows a pre-detrmined structure, but may contain metadata that is different from one object to another.

A JSON document can be tokenized into several parts, defined next, and these tokens can be used to track the position within the document while parsing.

Definition 2.1 (JSON token). is either one of the control characters, ‘{’ , ‘}’ , ‘[’ , and ‘]’ , a key, which is a string that comes before a colon, or a primitive value, e.g. a number or a string.

While parsing a JSON document similar to how it is done in [26], a stack is used to track the position. Encountering one of the open control characters { or [or a key results in a push operation. Encountering one of the closing characters } or] or reaching the end of a value after a key results in a pop operation, and it must correspond to the top of the stack.

A simple way to query a JSON structure is by using a JSONPath query [23], which is similar to the XPath [17] queries for XML data. Similar to XML, a valid JSON file is represented as one tree with a single root. This makes it unsuitable for SQL processing which requires a set of records, unless we are processing small JSON files, where each file represents one record.

Some JSON data can get very complex, with records containing hundreds of key-value pairs, and deeply nested objects. Furthermore, some queries can further complicate the parsing with the addition of filtering and descendant elements. To mitigate this, JPStream [26] introduces the concept of streaming-automaton that is built from a user provided JSONPath, and uses it to track the parsing state.

3 PROJECTION TREE

This section gives an overview of the projection tree, which is a data structure that dsJSON introduces to selectively parse JSON data and convert the extracted semi-structured JSON records to a structured data format. This is a vital step to run any SQL query on the JSON data and what distinguishes dsJSON from regular JSON parsers that just produce a JSON object. In relational algebra, projection is the operator that can add or remove columns in relations. The name of the projection tree comes from the fact that it selects a subset of values from different nesting levels from complex JSON documents, and converts them from a semi-structured textual format to a structured one that can be processed using relational algebra operators. Before formally defining the projection tree, first we provide definitions for its basic building blocks.

Definition 3.1 (JSON path). A set of JSON tokens that determine the position of a value within a JSON structure. These tokens contain open object '`{`', open array '`[`', or a key, i.e., attribute name.

For example, in Figure 1, both attributes `id` and `regularPrice` exist under the JSON path: ⟨ '`{`', '`products`', '`[`', '`{`' ⟩. Knowing the path leading to a value allows us to determine whether its value is needed, leading us to the *essential path*.

Definition 3.2 (Essential path). A path in a JSON document that leads to an attribute required for parsing a desired record.

In dsJSON, essential paths are determined from user provided queries, i.e., JSONPath and SQL queries. When parsing, a JSON path that does not correspond to an essential path triggers a skipping mechanism to discard values descending from it, until reaching the prefix of an essential path. To track all of these transitions and more, we unify them under the concept of the projection tree.

Definition 3.3 (Projection tree). A tree comprised of all essential paths that are used to selectively parse a JSON document and apply complex nested record projections. Each node in the tree corresponds to a token within an input JSON document, where leaf nodes correspond to primitive attribute values, e.g., numbers or strings. Transitions in the tree occur as JSON tokens are read from the input, and can trigger several operations like filtering, skipping, or projecting the selectively parsed values to a structured format.

The design of the projection tree makes it more efficient to overcome the limitations we are addressing. dsJSON goes through a few stages prior to producing the final structured output, and all stages relate to the projection tree. Figure 2 highlights the five stages of projection tree processing in dsJSON, namely, initial tree construction, input partitioning, schema inference, SQL push down, and row parsing, further detailed below.

The first stage in dsJSON is building an initial projection tree by processing the user-provided JSONPath queries into tree structures, and merging them. Figure 3 shows an example of a projection tree. The first part of the tree (nodes #0-#10) is built after processing the user provided JSONPath queries. This initial tree can only support sequential JSON parsing but cannot produce structured records. This process is detailed in Section 4.

To be able to process the input data in parallel, the next stage in dsJSON is partitioning. This stage uses the projection tree to determine how to shift the starting position at each partition, such that the starting JSON path at an arbitrary starting position is known, and to avoid splitting records among different processors. We provide two partitioning methods, which are specifically required for data in the general JSON format. In this stage, one node in the tree is identified as the *split node*, e.g., node #3 in Figure 3. Section 5 provides the details about this stage.

Next, schema inference is performed on the initial projection tree. This stage produces the expanded projection tree, adding nodes #11-#24 in the tree in Figure 3. This makes it possible

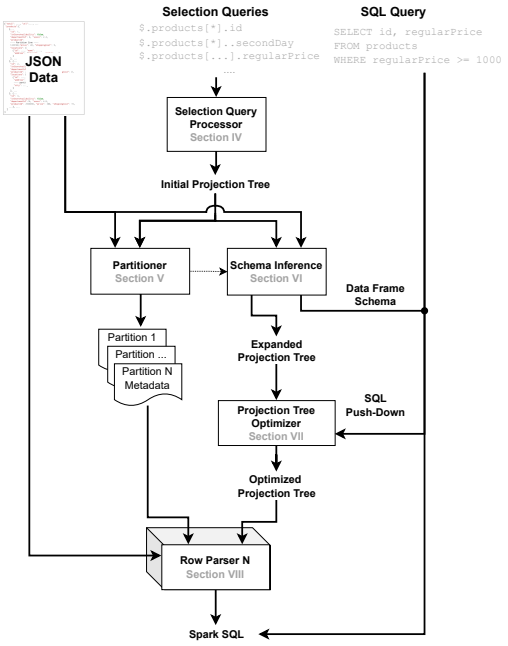


Fig. 2. System Architecture of dsJSON

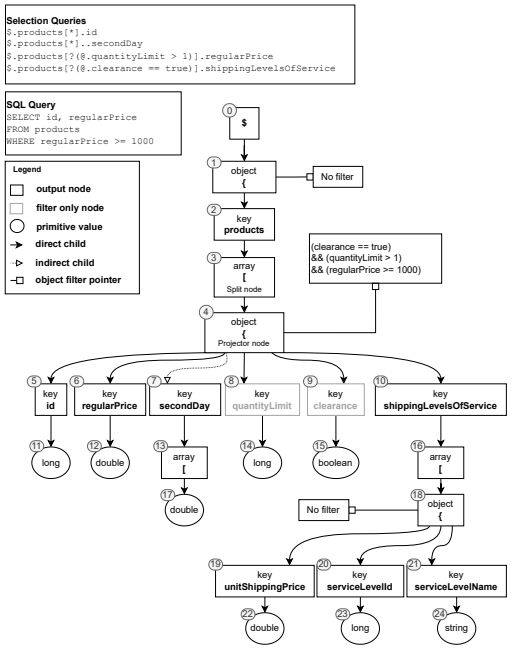


Fig. 3. Complete Projection Tree Example

to identify a structured format for the final output. Performing schema inference through the projection tree resolves some of the limitations in existing systems that cannot produce structured records. The details about schema inference are provided in Section 6.

The fourth stage is SQL push down that further optimizes the projection tree by pushing down SQL projection and filters. Based on the types of queries the user executes on the structured data, SparkSQL determines the required attributes and pushes down some of the filters, especially those non-aggregate in nature. In Figure 3, SQL push-down results in pruning nodes #10, #16, #18-#24, and adding the last predicate to the filter associated with node #4. In a lot of cases, SQL push-down may result in pruning or skipping hundreds of attributes which can provide significant performance gains. This step produces an optimized projection tree, which is what is used in the final stage of actually parsing the input records. This part is detailed in Section 7.

Finally, after obtaining the optimized version of the projection tree, the row parser starts producing the final output records in the expected schema. The transitions that occur in the projection tree while parsing JSON documents are detailed in Section 8.

4 JSONPATH QUERY PROCESSOR

The first problem we face when parsing JSON files is how to define the output records. Keep in mind that a well-formed JSON file is just one big object but it also contains many nested objects at various levels. There is really no right or wrong definition of which records to produce since it all depends on the user application. Notice that in the JSONLine format, each line in the input corresponds to one record in the output, while for the GeoJSON format each record is defined by a geometric feature. However, even for these formats it can be more practical to parse complex records more selectively. To resolve the issue of defining the selected attributes while giving the

user the full flexibility, we adopt the syntax of the JSONPath query as a method of defining which records to return.

In the remainder of this section, we first describe how a single JSONPath query is processed to construct a projection tree. Then, we show how multiple projection trees produced from different JSONPath queries are merged into a single projection tree. After that, we describe how JSONPath filters for each object are combined.

4.1 Processing a single JSONPath query

This part explains how we construct an initial projection tree from a single JSONPath query. A JSONPath query indicates to the parser that all values descending from this path must be included in the output. The first step in processing a JSONPath query is tokenizing it into its components, and identifying the type of each one. The first token in a query based on the JSONPath syntax always starts with the root symbol `$`. Following tokens can then be one of three types: *key-token*, *descendent-key-token*, and *array-token*.

The *key-token* type corresponds to a key in the corresponding depth in the JSON file, and is preceded by a dot, e.g., `'.products'`. The second token type is *descendent-key-token* written as `..` and followed by a key, which indicates that the depth of the JSON path for matching the next token should not be dependent on the corresponding level on the query, e.g., `'..secondDay'`. The third type is *array-token*, written as a pair of square brackets, e.g., `'[*]'`.

An array-token can can either take the wild card symbol `*` for accepting all elements in the array, or can take a filter, e.g., `'[?(@.categoryId==11)]'`. The original JSONPath query syntax can take the indexes of arrays as filters. However, we omit indexing since it is impractical to know the exact index of each object when the file is partitioned and parsed in parallel. This is because communication is not allowed among Spark executors, following the BSP model. For most use cases, filters and projections are sufficient for selecting the relevant records and the order of records in an array is not important. Furthermore, if a fixed number of records is desired, the parsing can stop after producing that desired number. Additionally, in most cases, the order is still preserved knowing the partition id, and we also provide the option to store an additional field that can be used to access the rows based on their order.

The original JSONPath filters are only supported for arrays, but for additional flexibility, we allow filters to be provided after any key in the query as long as the value is a nested JSON object, and it can be written between a pair of round parentheses instead of square brackets to differentiate between the different token types. For example, this makes it possible to add a filter on the `address` object, like '`$.products[*]..address(?(@.city == Seatle))`'.

After tokenizing a selection query and extracting any filters, those tokens are then used to build an initial projection tree. We show four examples in Figure 4. Notice that each token in a provided query has one corresponding node in the produced tree. Additionally, prior to any node corresponding to a `key-token` we add a preceding object node. Keys in a JSON document always exist within a JSON object. Also, object nodes in a projection tree are used to apply filters and transitioning down to and up from an object node corresponds to reading an `{` and `}`, respectively. The projection tree transitions are described in more detail in Section 8.

When building a projection tree, we identify a special node that determines when a record is fully processed and sent to the final output. We refer to this node as the projector node.

Definition 4.1 (Projector node). A node that indicates the level at which the record is complete and ready to return. It is always the first node from the root where the tree diverges into multiple paths.

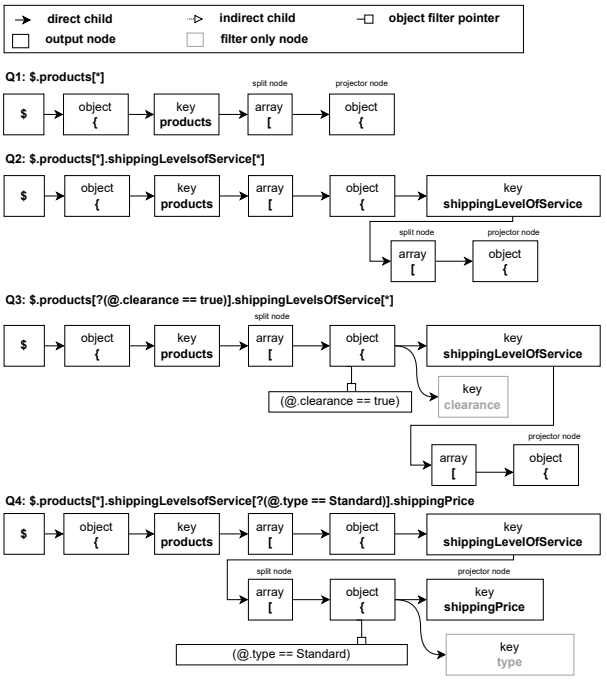


Fig. 4. Example projection trees from single JSONPaths

In a tree built from a single JSONPath, the projector node is always the last node corresponding to the last token in the query.

4.2 Merging multiple JSONPath queries

While one JSONPath query could be enough to properly select the desired set of records, there are many cases where the user might need to specify two or more JSONPath queries. For example, in Figure 1, the nested object contains hundreds of attributes but the user might want to select only '`serviceLevelId`' and '`unitShippingPrice`'. If we use a single JSONPath query, then all the sub-attributes will be selected which incur a huge overhead on all the stages of JSON parsing. Therefore, the design of dsJSON allows users to provide many JSONPath queries, which unlocks the potential for complex use cases.

The main part in merging multiple JSONPath queries is to find the most common path among them, ignoring any filters. The queries in Figure 3 all share the JSON path ⟨ '`{`', '`products`', '`[`', '`{`' ⟩. This corresponds to the first five nodes in their individual initial projection trees. Merging the queries results in producing an initial projection tree containing nodes #0-#10 shown in Figure 3. The last node that is common among all the queries (node #5) is defined as the projector node, since it is the first node when the initial trees start to diverge into multiple paths.

4.3 Defining object filters

The filter expressions, from the JSONPath queries, can be used to improve the performance of JSON parsing by early skipping records that do not match without fully parsing them. To integrate these filters into the projection tree, each filter is converted to an internal tree structure, e.g., Figure 5, and attached to corresponding nodes in the tree. The leaf nodes in the tree correspond to JSON

attributes (variables) or constants. Each internal node represents a predicate, e.g., ≥, or a Boolean operation, e.g., AND (&&). Additionally, we add a hashmap that maps each variable to its leaf node in the tree to quickly locate it. As soon as a variable is parsed from the input, we plug its value in the tree and propagate the evaluation up to the root. A predicate is evaluated when all its operands are available. For example, in Figure 5, since the root node is an AND operator, as soon as one of its children evaluates to `false`, the remainder of the record is skipped. In the case of the `AND` operator, it propagates if one of the operands evaluates to `false` or all the operands evaluate to `true`, while for the `OR` operator it propagates if one is `true` or all are `false`. By default, when a filter does not fully evaluate because a value is missing is considered `true`, the user can change this behavior by including a predicate that checks that these required values exist (`?(... && @.someAttribute != null)`). When merging multiple JSONPaths, the filters at corresponding nodes are also merged. While we build the initial projection tree using multiple JSONPath queries, we opted for the option to support only one, possibly compound, filter per object. By default, if an object has filters coming from multiple paths, they are merged using an `AND` operator. While it can be trivial to add support for multiple filters, and only discard values descending from the path of a filter that evaluates to `false`, it makes the behavior harder to follow from the perspective of the user. The user can control logical operators by simply writing the filter corresponding to a specific node in one JSONPath query, instead of splitting among multiple queries.

In addition to the simple filters on primitive attributes, dsJSON is extensible and can supports user-defined complex filters for different data types. In the experiments, we use filters for the Geometry data type when parsing GeoJSON files.

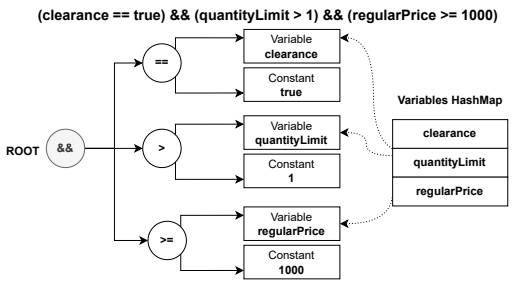


Fig. 5. Example filter expression tree and variables map

5 PARTITIONING

Similar to most big-data systems, Spark uses the bulk synchronous parallel (BSP) model in which asynchronous communication is not allowed between executors. This means that each partition in a JSON file should be parsed independently. However, the projection tree used to track the parsing will not be able to parse the second and subsequent partitions without being initialized at the correct node. This is very challenging since the initial path at an arbitrary position generally depends on the entire file before. The JSONLines format does not require this part, since the start and end of each partition is shifted to the next newline character, hence, the entire JSON object is always contained in only one partition.

To make sure that partitioning does not split output records and projection filters among two processors, the projection tree has a special node referred to as the *split node*.

Definition 5.1 (Split node). It is the deepest node in the tree prior to the projector node and any object with a filter. It determines the initial JSON path allowed at the start of a partition. The start

of a partition must be shifted such that its start is not descending from the path corresponding to this node to avoid splitting output records and filters between two partitions.

The *split node* in Figure 3 is node #3. When initializing the projection tree at an arbitrary token the partitioner must shift to a token at a path that is not descending from the split node. To illustrate this concept, Figure 6 shows how the positions of characters suitable for splitting change with different queries. The projection tree of each of these queries are those in Figure 4. Q1 extracts all objects in the products array. Each of these objects will represent one row in the final Dataframe (table) output. The split node, being at the array-node just prior to the projector node, makes any control character `{ } [ ] ,` or white space characters that are not within the matched objects suitable for splitting. Q2 selects inner objects, and this allows for more split points, as long as the matched inner objects are not split. Q3 is similar to Q2 except that it adds a filter to objects in the products array. This filter causes the split node to be the same as that of Q1, to ensure that filters are not split among partitions, and similarly for Q4.

This design choice has considerable practical benefits, but it imposes two minor limitations. First, it cannot efficiently partition a document when filters are added at the root node or with attributes that exist only at the start or end of a very large document. These types of filters might be useful to prune JSON documents out of a very large set. This will cause no issues for small files since they do not need to be partitioned, but for large documents, it hinders the level of parallelism that can be achieved. In practice, this is not a big issue since existing JSONPath processors only support applying filters to JSON arrays of type object, because typically users are interested in filtering large arrays. Furthermore, for this type of analysis, each matched row is not expected to be big. Second, it limits partitioning opportunities when matching documents that are extremely large. However, most database stores impose a limitation on the size of a row, usually in the order of megabytes, and this is still too small to cause any serious partitioning issues.

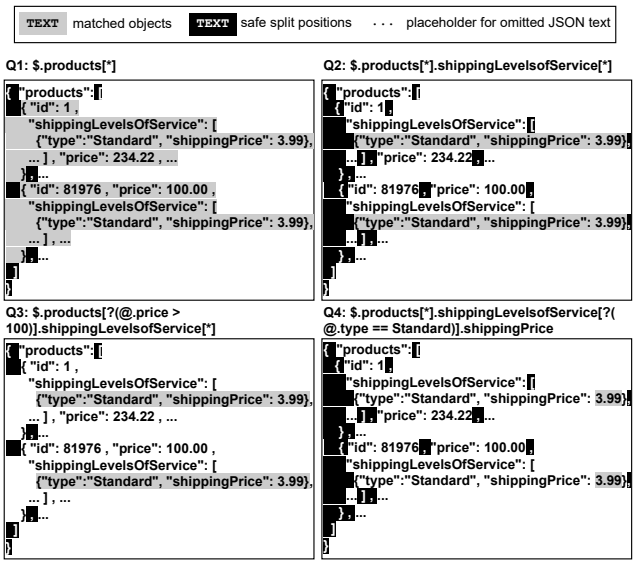


Fig. 6. Changes in split positions and matched records

To overcome this challenge, dsJSON provides two methods for partition initialization, *speculative* and *full pass*. The speculative method collects information from the start of the file and then

speculates based on this information at each partition. The second method uses a full pass on the file that tracks the path at the end of each partition. Then, a merge step determines exactly the start state of each partition. The full pass method is helpful in the uncommon use cases where speculation might fail. At the end of the partitioning method, the initial path for each partition is determined and is used to initialize the state in the row parsing stage.

```
{"total": ..., "url":..., ..., "products":[
  {..., "id": 1, "productId":
  ---- Partition line ----
  1435108,"locations": [
      {"id": ..., "name": ...,
       "address": {"street": ..., "city": ...} }
      ], ...},
  {..., "id": 2, "productId": 2536543,"locations":
  [ {"id": ..., "name": ...., "address": {"street": ...,
  ---- Partition line ---
  "city": ...} } ], ... }, ... ]}
```

Fig. 7. Partitioned JSON Data Example

5.1 Efficient with speculation

The key idea of the speculative partitioning method is to collect some information from the beginning of the file and use this information to accurately speculate the projection tree at each partition. This method assumes that the entire file will mostly follow the structure of the first part and will use that to resolve any confusion. For example, if it knows from the first part of the file that the key '`address`' appears only while the parser is at a specific projection tree node, the parser will use this information to adjust the projection tree node whenever that key is encountered.

This method collects the information needed for speculative partitioning only from a small portion at the start of the input file. We empirically found that in the majority of use cases, the first megabyte of the file is sufficient for accurate state identification. It starts from the beginning of the input file and collects all the encountered keys along with the corresponding path encountered prior to reaching the key. The key-collector also has a counter for how many times a key is encountered at a given path. The counter helps select the best keys for speculation when partitioning. To illustrate this, the key collector will generate the map in Table 1 from the JSON data in Figure 7. Some rows are omitted from space.

Table 1. Collected keys map

Keys	Path	Count
`total, products`	`{`	1
`id, locations`	`{ products [ {`	1000
`id, name, address`	`{ products [ { locations [ {`	5234
`city`	`{ products [ { locations [ { address {`	5234

This table is used to resolve the ambiguity of the starting node of the projection tree without having to start parsing from the beginning of the file. In most cases, the format of the JSON file is consistent throughout, which makes the parsing state similar when one of the keys in the table is encountered. Furthermore, most keys in the table have only one entry which makes it possible to identify the initial node without any ambiguity. It is also possible to speculate on a key that

has multiple entries as long as they result in the same projection tree node and exist at the same depth in the JSON structure and only differ in the value of the keys. However, there are cases where the same key could appear at multiple paths. For example, the key `id` in the table appears at two different paths. Moreover, the counter keeps track of the frequency of each occurrence so that the state initializer will rely more on the more frequent occurrences that has one entry in the table. The most frequent keys with only one entry, in this example, are `name`, `address`, and `city`, followed by `locations`. All of these make good choices for speculation. In the real dataset that this is based on, there are hundreds of such keys. If one of these keys is encountered, it is used to determine the state. In the rare case, where the matched key exists at a different path than the one on the table, then the wrong node will be identified.

Based on this concept, the speculative partitioning method continues after building the table in two steps. First, it defines an initial data-oblivious partitioning that simply partitions the file into fixed-size partitions, e.g., 128 MB each. Second, it applies a data-aware adjustment step that shifts the partition boundaries to the next identified key from the table. Notice that the adjustment step happens in parallel since each partition boundary is shifted independently, which suits the BSP model. After that, the position is shifted to match a node at or prior to the split node to ensure that a matching record is not split. Speculation can also be improved by searching for multiple keys that are correlated, even if some of them exist in multiple paths. However, since we found that practically the simpler version works well we leave this as a future improvement.

In the example in Table 1, if the partition initially starts at the `name` field, which is at a path that is descending from the path corresponding to the split node, then it must search for the following characters `}`, `]`, and `}`, respectively, to arrive at the split node. This ensures that matching records are not split.

With invalid speculation, the parser either will encounter unmatched closing characters when traversing the projection tree, which would raise an exception, or the parser will continue unaware of the error. However, if the parser continues, it is unlikely that it will match any records, or it will match records that look very different from the inferred schema. To ensure that no errors happen, we add a post-processing check that verifies that the initial path of each partition matches the termination path of the previous partition. If they are equal for all consecutive partitions, then speculation is guaranteed to be correct. The program raises an exception if speculation is not verified with information about at which partition it occurred and the token that was used. Users will then fallback to the full-pass method which is described next.

5.2 Exact with a full file pass

While speculation works most of the time, there are cases where it could fail. It is when the schema of the first N records is not enough to generalize to all records in the file. This can happen when a dataset contains recursive objects. Consider the `products` array example shown earlier, if records within it also contain an attribute called `relatedProducts` that also contains all the attributes within the parent product, it would not be possible to speculate on this dataset since for any attribute within the `products` array there are at least two possible parsing states, assuming the `relatedProducts` is also repeated and the nesting can go multiple levels deep. In this case, the only option is to do the partitioning using the full-pass method. The user can also try to normalize the structure of the data through appropriate JSONPath queries by only selecting the `productId` attribute of all related products, for example.

One reasonable use case that we were able to identify is partitioning GeoJSON files that contain nested GeometryCollections. That is, it contains objects of type GeometryCollection that also contain GeometryCollection objects within them, and the nesting can go multiple levels deep. This way, all the keys in the file will exist in multiple depths. Using speculation on this file, will raise an

exception, since filtering the encountered-keys table will result only in the keys that exist once at the top of the file. Hence, a full pass partitioner is required for these use cases that rarely occur in practice. Previous work considers multiple paths that are possible and start parsing assuming one of them, and restarting when a path is identified as false. However, the way the system is designed based on the Spark Data Source API, records are returned as soon as they are parsed. While we can delay returning those values until we eliminate all the alternative possibilities, which might result in multiple full passes over a partition, we opted to have one full pass that guarantees the correct state is initialized, and ensure no significant memory overhead is required to store intermediate values. The data is not fully parsed during this iteration, and it requires minimal memory and communication overhead since only a small array is gathered from all the partitions at the end of this stage.

The key idea of the full-pass partitioner is to scan all partitions in parallel and record the path that leads to each key *within this partition*. Except for the first partition, the second and subsequent partitions will record invalid paths since they are unaware of the *initial path* at the beginning of this partition. For example, in Figure 7, the second partition starts at the path ⟨ '{', 'products', '[', '{', 'productId' ⟩. To resolve this issue, the full-pass partitioner keeps track of two additional pieces of information for each partition. 1) The *inner path*, which is the path that leads to the end of the partition that is not closed yet as shown in Table 2. 2) Any close object '}' or close array ']' tokens that are not matched with a corresponding open token. This information is collected at a central node that corrects the paths within each partition. The idea is to propagate the inner paths, representing open token with no closing token, with non-matching close token in the same order of partitions to determine the initial path for each partition. Once the initial path is determined, the correct path leading to each token can be easily calculated by prepending the initial path to the one computed within the partition. Finally, the start and end position of each partition are shifted according to the stored positions of the latest open symbol. Using this method also makes it possible to skip an entire partition in later processing if it starts at a redundant path that does not end by the end of the partition. The start positions are also shifted to match the split node similar to the previous method.

The main edge case when applying the full-pass partitioning function is when the start of a partition falls within a string. This is easily handled knowing the JSON syntax. Simply, we start parsing as if the partition does not start in a string. If parsing fails due to incorrect JSON format, we restart assuming that the partition starts within a string. Interested readers can refer to [25] for more details.

Table 2. Full-pass partitioning complete output

Part	Inner Path	Initial Path
0	{ products [{ productId	EMPTY
1	} { locations [{ address {	{ products [{ productId
2	} }] }] }	{ products [{ locations [{ address {

6 SCHEMA INFERENCE

This step takes as input the initial projection tree and the input data. It produces a unified schema for all records that will be added to the final output. The schema is defined as a list of pairs (name, type), each indicating an attribute with the name that appears in the JSON file and its data type, e.g., string, numeric or a nested structure. This schema is important for SparkSQL which needs an initial schema to be able to parse and analyze the SQL queries that the users provide. For example, it needs to verify that there is a numeric column named `regularPrice` for the SQL query in Figure 3. This

Table 3. Inferred schema after expanding projection tree

id	regularPrice	secondDay	shippingLevelsOfService		
			array[object]		
			unitShippingPrice	secondLevelId	serviceLevelName
long	double	array[double]	double	long	string

section describes two methods for schema inference, optimistic and pessimistic. Then, it follows with a discussion on how to mitigate a JSON file with an extremely complex schema.

The optimistic method produces the schema from a fixed size at the start of the input data, which we limit to 1000 records by default, but it is left as an option to the user. This method is much more efficient since it uses a very small portion of the data under the realistic assumption that all other records will follow the same schema. The likelihood that the matched records follow the exact same schema increases based on how selective the JSONPath filters provided by the user. For example, given a filter that selects products from a specific category, then, the variability of the schema would be minimal, hence, only a small number of records will be needed to build a unified schema. The pessimistic method runs a full pass over the file to ensure the result is correct without making assumptions.

Optimistic schema inference works by first getting the initial projection tree in order to find matching records. It starts parsing from the beginning of the file, whenever a record is matched with the JSONPath query, it stores all attribute names and their inferred data types. Figure 3 illustrates how schema inference works. In a sense, dsJSON expands the initial projection tree (nodes #0-#10) when parsing a record, producing all other nodes in the tree. After it completes a record, the row projector node returns the row schema of the matched record, like the one shown in Table 3. Then, as more records are matched, the schema of all matched records are merged into one while updating data types to be more broad. Most type conflicts are when all previously obtained records have a key of undefined type, while a new record might have a different type. Some attribute types might also be resolved from long type to a double type, as new records are matched. We limit the inference to be based on at most the first 1000 records by default. We found that this default number produces accurate schema while still being reasonably efficient, but it can be easily changed by the user. This method is executed simultaneously with the partitioning stage if the user chose the speculation method.

For the *pessimistic schema inference*, first it is required that the data is partitioned and the projection tree at each partition is initialized accurately, so that this stage is processed in parallel. Pessimistic schema inference is required for data where some keys may only exist in some records, and when the entire correct schema is required. It works similar to the optimistic schema inference described above, but it uses the entire data. After each partition is processed independently similar to the optimistic approach to produce a schema. Then, all schemata are merged to produce the final complete schema.

Extremely complex schema: In some use cases, merging the schema of matched records may result in a very large nested schema that causes failures due to requiring a very large memory footprint. This is because some fields may not have uniform nested structures across the data. In existing systems, like in the Apache Spark JSON reader, failures can only be mitigated when the user manually provides the schema and setting those complex nested objects as string columns, or discarding them completely. However, the data in these columns would still be in their raw semi-structured JSON format, and will not benefit readily from the existing tools like SparkSQL queries without additional processing iterations that might require implementing some custom

functionality. dsJSON resolves this schema expansion issue in three ways, filters, multiple JSONPath queries, and lazy parsing.

First, because dsJSON supports applying filters when inferring the schema, providing appropriate filters results in selecting objects that are more uniform in structure, minimizing the issue of schema expansion. Consider the OpenStreetMap dataset [40] that we use in the experiments, without using filters, the merged schema grows very large resulting in an estimated size of 1MB per record, which is much larger than the size of a record individually, eventually resulting in execution failure due to running out of memory. This is because it has a large variety of objects which have different attributes in their metadata. Adding appropriate filters, like limiting the selected types and geometric boundaries, reduces the estimated size per record to 400 bytes, which is the expected size for 20 attributes of type string, thus, avoiding the failure. This is not to mention that this dataset cannot be processed using the existing Spark JSON reader, because it is not in the JSONLines format, and the entire dataset is one large JSON object that exceeds two terabytes in size. While Beast [21] supports reading GeoJSON files, its current version fails while partitioning this dataset, moreover, it doesn't support applying analytical queries like those provided by Spark SQL, because it doesn't have schema inference. Therefore, this only leaves dsJSON as the only solution that supports this type of complex large scale data processing.

Secondly, because dsJSON supports extracting values using multiple JSONPath queries, this allows designing path queries that avoid sections in the JSON data that causes schema expansion. The Wikipedia dataset [8] that is also used in the experiments causes schema expansion failure when the schema of the entire records are merged. The way this dataset is designed includes id names as keys within objects, and the size of one record can be hundreds of kilobytes. When merging the schema of different objects, the schema size keeps increasing until the program fails. dsJSON avoids this issue by using multiple JSONPath queries including descending paths, which results in building a compact schema that covers only the relevant part of the data for the desired output.

In this case, the user can adjust their JSONPath queries in a way such that that the final matched records are more unified, especially with the use of descendent attributes. There is also the option to apply the JSONPath filters while building the schema. An appropriate filter may lead to selecting records that mostly follow the same schema, thus avoiding that issue. Without these options, it would not be possible to find a schema of a reasonable size that matches all the records. For example, because none of these options is available in the JSONLines reader in Apache Spark, it fails prematurely while inferring the schema of some files, as shown later in the experiments.

For the lazy parsing method, dsJSON provides the option to automatically detect fields that may result in schema expansion by setting a threshold for the number of allowed subfields, and treat those detected fields as strings, where they can still be processed separately in later stages and they will need to be parsed then. This comes as the last option for avoiding schema expansion without any loss of data. The user also has the option to provide their own schema which would save the schema inference time, assuming the provided schema is valid for the matched records.

7 PROJECTION TREE OPTIMIZER

This section shows how the projection tree is finalized by pushingSQL filters and projections down into the tree. While we can simply let SparkSQL operators apply the projection and filter operations, this additional optimization step can greatly speed up the parsing by skipping parts of the input that are not needed. For example, if the filter does not pass, the rest of the record can be skipped. Also, if only a few attributes are needed, the parsing can stop once these attributes are found.

Note that SQL queries can only be applied to the fields in the inferred schema of the records matching the JSONPath queries, since Spark SQL [10] requires analyzing the schema before pushing down filters and projections, and the schema is based only on the output of the matched records.

SQL projection push-down optimizes the projection tree by pruning paths that lead to non-required attributes. SparkSQL pushes-down projections as a list of fields in the schema that are required for the applied analysis. Based on this list, different changes on the projection tree are applied. If a node does not exist on the list of required attributes and it is not used in a filter, then it is pruned out of the tree along with its descendent nodes. If a node does not exist on the list but it is used in a filter, it is changed to a filter-only node. Nodes that exist on the list are kept as-is. In the example provided in Figure 3, this operation results in pruning nodes #6, #7 and #10 and all of their descending nodes. In practice, as further confirmed in the experiments, projection push-down can result in pruning hundreds of nodes, which provides significant performance gains.

SQL filter push-down is integrated by converting the provided filter to an internal expression and appending it to the filter associated with the *projector node*, e.g., node #4 in Figure 3. It also results in updating the expression tree of the associated filter, like the one in Figure 5. Note that the existing JSON reader in Spark also implements projection and filter push-down which confirms the effectiveness of this step. However, dsJSON makes two fundamental differences. First, Spark uses the Jackson parser [5] as a black-box and adds the projection and filter support as a post-processing step while dsJSON integrates them into the row parsing. Second, dsJSON seamlessly combines JSONPath and SQL projections and filters into one data structure, i.e., the projection tree, which makes it more powerful than Spark that only works with SQL filters.

8 ROW PARSER

In this final stage, each worker in a distributed cluster takes one partition produced by the partitioner, i.e., start and end offsets in the file, as well as, the initial JSON path. It initializes the optimized projection tree accordingly. The first partition always starts at an empty path and is initialized to the root node. Other partitions are initialized as discussed in Section 5 so that the active node matches the initial path. Then, it iterates over the input, token-by-token, while processing them using the projection tree. When the projection tree transitions into the projector node, it starts parsing a new row. This section first describes the type of operations that occur when processing a projection tree, and when they are triggered. Then, we describe the error handling mechanism.

8.1 Projection Tree Operations

Several operations can occur as the parser iterates over the input.

Transition downward. It occurs when the parser reads a token that matches the token of one of the children of the current active node. Object nodes are matched with an open curly-brace (`{`), array nodes are matched with an open square-bracket (`[`), key nodes are matched with an equivalent key of the node, while primitive values are matched with a primitive value of the same type.

Transition upward. This occurs when the parser completes reading a value corresponding to a node. For objects, it occurs after reading the closing curly-brace (`}`). For arrays, it is after reading the closing square-bracket (`]`). For key nodes, it occurs when the value associated with the key is fully processed, and results in adding it to its parent object. An upward transition from any value results in adding it to its parent node or appending it to its parent array. Additionally, an upward transition from the projector node triggers returning the parsed record to the output, and resets filters.

Propagate to filter. A value is propagated to a filter after an upward transition from a key node, if the key is one of the variables in the filter expression tree for its parent object. If the filter

evaluates, to `true` or is still not fully evaluated, processing continues normally. Conversely, if the filter evaluates to `false` the remainder of the object is skipped, and any already processed values descending from this object are discarded.

Skip redundant paths. This operation is triggered when there is no matching essential path for a downward transition, resulting in skipping a value without serializing it. The active node remains the same if its value is not yet fully processed. Skipping is also triggered at object nodes when all of the required values are obtained, or a filter is evaluated to `false`. It searches for the matching closing character keeping track of all control character in between.

These summarize the main operations that occur when processing a JSON document using a projection tree.

8.2 Error Handling

The flexibility of JSON means that unexpected input tokens are bound to be encountered, especially when dealing with very large scale data coming from different sources. Most existing JSON parsers fail immediately when encountering unexpected tokens or malformed inputs. However, we designed dsJSON to work in a more permissive mode, making it a lot more robust to failure. Next, we discuss how some unexpected tokens are handled.

Handling mismatched types or invalid primitives. This occurs when the schema expects a specific type while the input contains a different one. In this case, it is easy to avoid any errors. First, dsJSON attempts to convert the type to the expected type. For example, if the expected type is a string and the encountered type is numeric, this number value is stored as a string. If the value cannot be converted to the expected type, it is simply stored as `null`. This possibility of data loss is avoided when using pessimistic schema inference, since it would ensure selecting the broadest datatype, and in case of mismatched types, it is by default set to a string.

Handling malformed JSON structures. Failures from some syntax errors can be easily avoided. For example, missing comma characters between array values, or two key-value pairs within an object, can be easily ignored. More problematic syntax errors are those related to control characters, the curly-braces, the square-brackets, and the double quote character. If an input file has a missing chunk at the end, then that can easily be ignored, and the parts processed prior to it are considered valid. However, we consider encountering non-matching closing tokens a fatal error, which happens when the last token in the current path corresponding to the active node in the projection node does not match it. This is because ignoring such a syntax error would put the execution in speculative mode, similar to what happens with invalid speculation. If it is important that malformed structures are detected, the user can use full-pass partitioning which also can be used to verify the structural integrity of JSON documents.

9 EXPERIMENTS

In this section, we execute an extensive set of experiments to provide an experimental evidence of the scalability and efficiency of dsJSON over existing JSON parsers. We also evaluate the effectiveness of the components that comprise dsJSON. Notice that since dsJSON supports some use cases that are not supported by any of the existing parsers, e.g., parsing general JSON files, there is sometimes no suitable baseline to compare to. We also provide a more detailed example to showcase one example enabled by dsJSON.

9.1 Experimental Setup

The experiments are executed on a cluster with one head node and twelve worker nodes. The head node is running on two Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz, with 8 cores per chip and 2 sockets per core for a total of 32 processing units. Each worker node is running on Intel(R) Xeon(R)

Table 4. Experiments datasets and queries

Dataset	Short name	Format	Size	JSONPath
Bestbuy [4]	BB	Std. JSON	1GB	`$.products[*]`
IMDB [15]	IMDB	Std. JSON	7.2GB	`[*]`
Wikipedia [8]	Wiki	JSONLine	1.3TB	`[*].claims..mainsnak`
MSBuildings [36]	MSB	GeoJSON	28.5GB	`$.features[*]`
OpenStreetMap [40]	OSM	GeoJSON	2TB	`$.features[*]`

CPU E5-2603 v4 @ 1.70GHz with two sockets, for a total of 12 processing units per node, resulting in a total of 144 processing units among all worker nodes. The head node is running on 128GB of RAM, while each data node is running on 64GB. The data is stored using Hadoop Distributed File System (HDFS) running on the same worker nodes. Also, Table 4 shows the datasets used along with the JSONPath queries. We use the same version of the BB dataset from [26] but we scale it to 16GB for all experiments, unless otherwise indicated.

We use the end-to-end running time as a performance metric which includes any required reformatting, parsing, and processing time. During experiments, we vary the input size and the SQL query that we run on each dataset. We compare to four sequential parsers and three distributed parsers. The sequential parsers are JPStream [26], SIMDJSON [32], Jayway [6], and Python JSONDecoder [1]. The distributed parsers are the built-in Spark JSONLine reader, a hand-crafted RDD-based Spark+Jayway reader, and Beast [21] which supports GeoJSON files. We added the hand-crafted parser to enable JSONPath processing for JSONLine files, a feature that is not supported by the built-in Spark JSON reader.

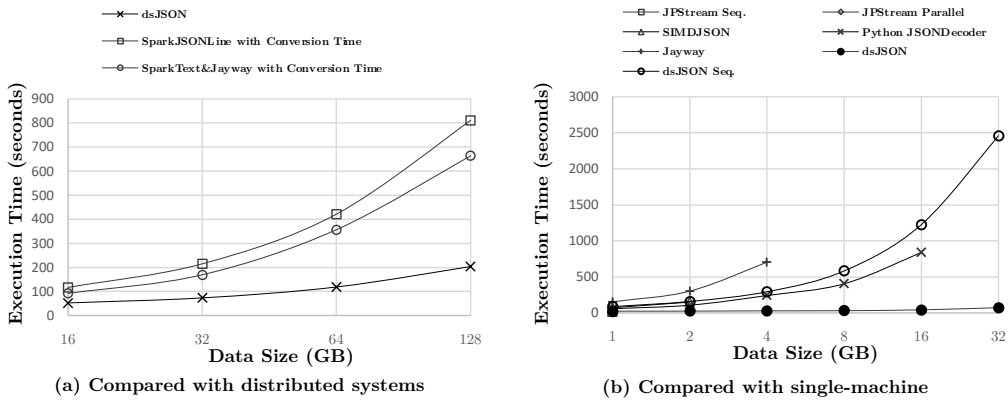
Table 5. Comparing to other distributed systems

System	Qualitative Comparison					Total execution time (seconds)				
	Std. JSON	JSON Line	Geo JSON	JSON Path	SQL	BB 16GB	IMDB 7.2GB	MSB 29GB	Wiki 1.3TB	OSM 2TB
dsJSON	✓	✓	✓	✓	✓	52	32	86	1275	2712
Spark JSON Reader		✓			✓	117 (70+47)	91 (60+31)	N/A	Fails	N/A
Spark Text Reader + Jayway		✓		✓		92 (70+22)	89 (60+29)	N/A	1222	N/A
BEAST			✓			N/A	N/A	42	N/A	Fails

9.2 Scalability of Distributed Parsing

Due to the variety of baselines and the different features supported by each, we show a qualitative comparison between these on the left half of Table 5. These missing features in other baselines limit the quantitative experiments that we can run. In this part of the experiments, when processing a standard JSON file, we run an efficient reformatting step that converts it to a JSONLine format that can be processed by Spark JSON and Spark+Jayway readers. We report the conversion time separately for convenience. The conversion scripts we used are custom to each dataset and only perform string operations without the need for full serialization which makes them highly efficient.

The right half of Table 5 reports the total execution time, where N/A indicates a non-supported case. When conversion is needed, the breakdown is provided between parentheses (conversion + processing). Firstly, only dsJSON can successfully process all the datasets and is consistently scalable. Spark JSON reader lags behind for both BB and IMDB datasets due to the bottleneck in file conversion. Interestingly, even after conversion, dsJSON is very close to the parsing step, which indicates the low overhead of schema inference and partitioning. This slight difference is mostly due to implementation details like the way the input reader is used and intermediate data



(a) Compared with distributed systems (b) Compared with single-machine

Fig. 8. Processing time with increasing input size

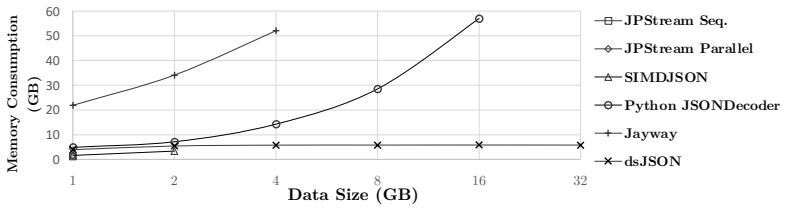


Fig. 9. Memory usage compared to single machine parsers

type conversion. However, given that this is a first cut implementation of dsJSON, the results are promising for moving forward to potentially replace the existing Spark JSON reader, after some development iterations.

For GeoJSON files (MSB and OSM), dsJSON can still parse them efficiently. For MSB, Beast was almost twice as fast for three reasons. First, it runs on the low-level RDD API, hence, does not require a schema inference step. Second, the code is hand-crafted to support only GeoJSON files while dsJSON is more general. Third, Beast can directly parse geometry objects into a compact in-memory representation while dsJSON keeps it as a nested JSON object with a complex schema. Even with that, Beast failed to parse the 2TB OSM file due to parsing errors. In the future, we can add a custom parser to GeoJSON files that can represent geometry objects in a compact way to reduce the schema size and increase efficiency.

One of the main motivating factors for introducing dsJSON is to skip the conversion step from the standard JSON format to the JSONLine format for very large datasets. Figure 8 shows the scalability of dsJSON when parsing standard JSON files with a `SELECT *` query. In this experiment, we scale the BB dataset from 16GB up to 128GB, and compare the time for converting and processing using existing systems against only using dsJSON.

Figure 8(a) compares dsJSON to the distributed parsers, Spark JSON reader and Spark+Jayway RDD parser,including the required time to convert to JSONLine. Since the conversion step requires a full pass over the file, dsJSON is up-to 400% faster. Note that the conversion step is IO-bound so a more efficient converter will not help much while dsJSON completely eliminates this step.

In Figure 8(b), we compare to four single-machine parsers which can parse and process a JSONPath query in a single scan over the data. JPStream [26] (both sequential and parallel) and SIMDJSON are implemented in C++, Jayway [6] is in Java, and the Python JSONDecoder is readily

Table 6. dsJSON breakdown of processing stages in seconds

Dataset	Schema	Partitioning	Parsing	Verification	Other	Total
BB	2	4	27	2.5	12.5	48
IMDB	1	4	14	0.5	12.5	32
MSB	2	4	60	1	12	79
Wiki	1	16	1538	5	20	1580
OSM	1	41	2938	10	15	3005

available in Python. All of them run on the head node. We use `$.products[*].categoryPath[*]` as the JSONPath query which selects records with only two attributes. For SIMDJSON, we use its interface to extract the same values. This query reduces the complexity and size of the final output, since some of these single-machine parsers are not optimized for complex output. All of these parsers produce an error as we increase the data size, mainly due to memory requirement. dsJSON scales well to large data sizes even in the single machine serial execution case.

We also show the memory consumption in Figure 9. The execution time and memory consumption are measured using the `time` tool in Linux. When measuring the memory consumption, we executed dsJSON in a single machine to make it easier to capture its total consumption. Also, note that memory allocation in Java, and Spark specifically, works differently. It allocates a large memory chunk for all Spark components as configured in the cluster. Therefore, this experiment focuses on how the memory grows as the input size increases. For the 1 GB file, JPStream executes the fastest and with the least memory consumption. However, its current implementation fails for larger sizes, due to implementation issues. We attempted to modify its implementation to make it scale to larger sizes, but it caused memory leaks and we conducted this finding to its authors. Regardless, its design requires storing the entire extracted data for filtering and finalizing the output. SIMDJSON only works on data less than 4GB, although it can be easily scaled on JSONLines, especially for queries that don't require aggregation of results. It consumes memory at a factor of 1.6 of the input size. The other two parsers are considerably slower and use considerably more memory. Except on the 1GB file, dsJSON executes the fastest, and its memory consumption scales well as data increases. The most important part is that dsJSON can process arbitrarily large files without requiring extra memory. To confirm that, we estimated the memory used by all the data structures in dsJSON using the `SizeEstimator` Spark library, and they only consume a few hundred kilobytes per process. Finally, when executing dsJSON in a single thread, we observed that SIMDJSON is about 60% faster, highlighting a major potential benefit of integrating them.

Finally, Table 6 shows the breakdown of all the stages of dsJSON. These results are for executing a `SELECT *` query with no filtering and with *speculative partitioning* and *optimistic schema inference*. The main takeaway is the low overhead of schema inference and partitioning steps even for the very large datasets, e.g., 31 seconds in the optimistic case for the 2TB one. Verification does not add considerable overhead and it passes for all datasets and queries.

9.3 SQL Integration

Following, we study the effect of pushing down filtering and projection on the parsing time. To study the effect of projection, we study two extreme SELECT clauses, `SELECT COUNT(*)` and `SELECT *` which project an empty set and a full set of attributes, respectively. Additionally, to study the effect of filtering, we study three `WHERE` clauses, no-filter, delayed filter, and early filter. *No-filter* does not apply any filters at all. *Delayed filter* applies a condition that select about 1% of the data and works on attributes that appear towards the end of each record in the file. *Early filter* applies a condition that also select 1% of the data but works on attributes that appear towards the beginning

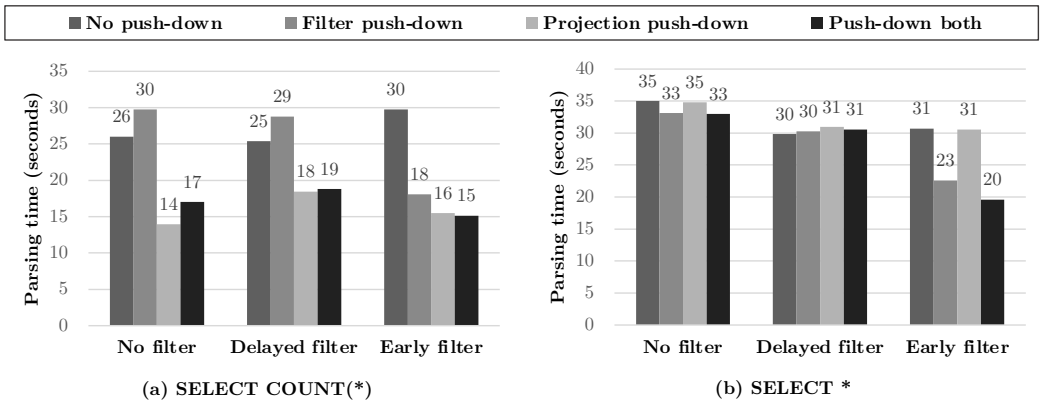


Fig. 10. Effect of integrating SQL on dsJSON

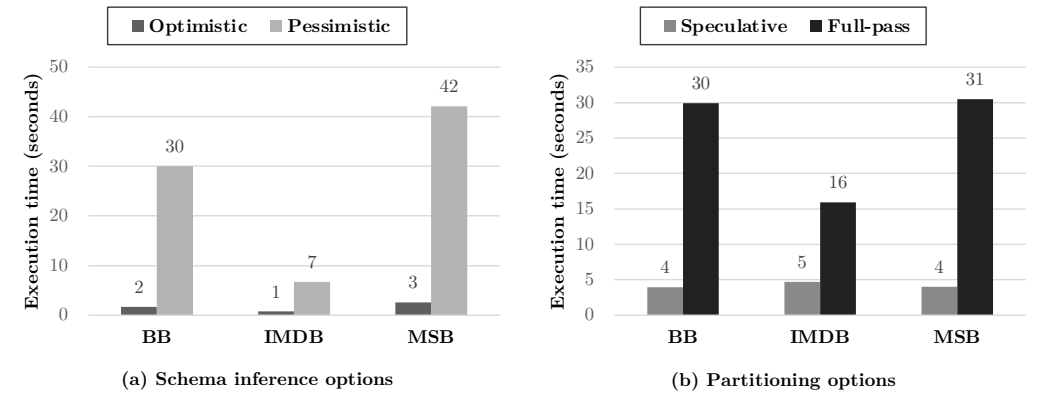


Fig. 11. Comparing partitioning and schema inference

of each record. With early filter, we expect the parser to be more efficient since it can skip parsing the rest of the record once the condition fails.

The results are shown in Figure 10. The times shown in this figure are the average of five executions, to reduce the effect of variability, especially due to scheduling, and fetching files from HDFS. These results support our claims in the paper. With a `COUNT(*)` query, projection push-down can save up-to 50% of the execution time. Similarly, with the early filter condition, the speedup of filter push-down is significant even with the `SELECT *` query. In the worst case, the filter and projection push-down add minimal or no overhead so it would be wise to enable these features by default.

9.4 Partitioning and Schema Inference

Next, we study the effect of the partitioning and schema inference techniques given different datasets. The results of this evaluation are shown in Figure 11. The optimistic schema inference depends on the average size of the matched records. Since for larger records, it would have to consume more bytes to build a schema on the first 1000 matches. It also depends on the complexity of the records and the level of nesting. The time for the optimistic schema inference also incorporates the time for the key-collector, since they are performed simultaneously. Partitioning with speculation is

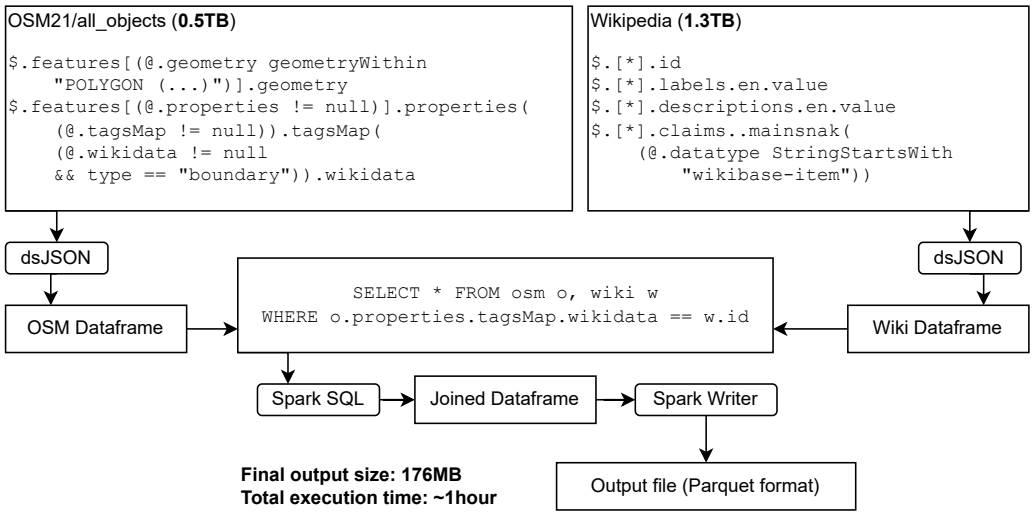


Fig. 12. Complete use case example

always small relative to the total execution time, and this time also involves issuing the tasks and collecting the results, since it is performed as a separate stage. Some of this time can be saved by making the speculation as the first step in the parsing stage, should that be desired. The pessimistic schema inference can be faster or slower than full-pass partitioning, depending on the nature of the schema. Some may contain nested objects and arrays, which takes more operations to merge than a flat schema. The combination of the optimistic schema inference and partitioning with speculation provide basis for the best-case, while the combination of the pessimistic schema inference and the full-pass methods provide basis for the worst-case.

9.5 Detailed Use Case

To demonstrate the power of dsJSON in running arbitrary SQL queries on JSON files, we perform a slightly complex SQL query, that involves join, on data extracted from the OSM and the Wiki datasets. The OSM version in this experiment is smaller in size, but contains records with more metadata. From OSM, we select records within the US of type `boundary` (usually a state, or county boundaries, etc.) that has a reference to a Wikipedia record. From Wiki, we select the id, the English label and description, and all the references to other Wiki items. Then, both dataframes are joined based on the Wiki id. The final dataframe contains the geometry objects of the selected OSM records as well as all references to Wikipedia items related to it. We then store this dataframe in Parquet, which is a column oriented data format, just to demonstrate the full integration with SparkSQL. Being able to process two very big JSON datasets by extracting the desired information and perform an expensive join operation on them in an hour is extremely valuable. Note that both of these datasets are poorly supported by existing parsers, as discussed in Section 6, so a user that needs to perform a similar operation in a distributed fashion might need to spend a lot of effort in implementing a custom processor specific to their use case. While this is a more memory intensive query, Spark successfully completes it with the same fixed memory allocated for each executor.

10 RELATED WORK

This section goes over the most related work to the contributions proposed in this paper. First, we compare dsJSON to existing parallel implementations and show how it differs based on the following attributes: A1) Can process data in the general JSON format in parallel, A2) Scales well

to very large data sizes, A3) Can directly apply SQL queries or other types of analysis, and A4) Provides options to handle complex JSON schema, as summarized in Table 7.

Table 7. Comparing dsJSON to parallel implementations

Processor	dsJSON	Spark [3]	AsterixDB [9]	BEAST [21]	SparkJayway	JPStream [26]	Pison [25]
A1	✓					✓	✓
A2	✓	✓	✓	✓			
A3	✓	✓	✓	✓	✓		
A4	✓						

Existing parallel JSON parsers [25, 26] don't scale well to large data, mainly due to their memory requirements and the fact they are not designed following the BSP model, making it difficult to adopt their proposed design in distributed systems, as discussed.

Existing distributed systems only support a specific subset of JSON like JSONLines [3, 9, 38] and GeoJSON [21]. To process very big JSON data in these systems, it can either be done sequentially or the user must convert it first to the JSONLines format sequentially or using a customized implementation. dsJSON supports all of these formats as well as the general format, eliminating the need for sequential conversion. dsJSON also solves the issue of complex attribute selection by inferring a schema using multiple JSONPath queries, making it possible to unify the structure of the matched records, in cases where it is not straightforward to unify them.

On top of not scaling to big data, most single machine implementation are not integrated with a data analytics systems, making it a challenge to perform even a simple type of analysis. dsJSON, however, supports this functionality out of the box, since it is fully integrated in Spark. Some prior work focused on analytics, including [14, 20, 24]. However, their focus is more on integrating JSON into existing RDBMS, but they don't tackle the issue of distributed processing of very large JSON files or schema inference.

Single machine environment parsers like [7, 16, 28, 32, 33, 37] focus on adding low level optimizations, like utilizing SIMD instructions. dsJSON on the other hand focuses on solving issues related to scalability and big data analysis. The optimizations they propose can be integrated into dsJSON in the future. SIMD optimizations can further improve dsJSON. The optimizations introduced in [28] can improve the skip functionality in dsJSON. The indexing techniques in [25] can be used to improve the full-pass partitioning, by utilizing SIMD instructions to track the positions of control characters. Furthermore, the optimizations in [32] can be used to improve the main parsing process and serializing the objects. There are several approaches for this integration: including using the SIMD supoort in the Vector API in Java, utilizing the Java-Native-Interface to communicate with C++ code, or implementing dsJSON in a low level language in a system that is compatible with the Dataframe API and the BSP model. There are many sequential JSON parsers, implemented in different languages with different features. Interested readers can refer to a JSON parsing benchmark [39] for details.

There are some similarities between parsing JSON and parsing XML, [26] goes through some of these differences and how working with JSON is more challenging. The work in [34] provides a parallel XML parser that divides the data based on the learned schema. It basically identifies the position in the structure at each partition based on the open tag of array elements, however, these open tags do not exist in JSON and this limits the splitting to array objects. In [27], they use the learned grammar of the XML data to reduce the possible execution paths at each partition. We opted for avoiding techniques that consider multiple execution paths to avoid having to fully parse records more than once, which may not scale well and also does not fit the computational model where results are immediately sent to the next processing stages. Moreover, [13] provides

a grammar based parallel parser that also supports JSON, but can only provide outputs once the entire data is parsed.

dsJSON takes distributed JSON parsing to a whole new level by providing a general-purpose parser that can support any JSON file. The work in [22] provides a parallel parser for the CSV format. Similar to our work, they provide a speculative approach and a fallback full-pass alternative, however, JSON requires more complex partitioning and schema inference. While we provide schema inference techniques, our focus was not on schema analysis and discussing the intricacies associated with it, refer to [11, 12, 18, 19] for details. [31] is similar in that it builds a data structure that stores the inferred types, but it focuses on finding discrepancies or updates in the schema. dsJSON instead infers a unifying schema based on attributes selected by the user as discussed earlier.

11 CONCLUSION

This paper introduced dsJSON, a full-featured scalable JSON processor for distributed shared-nothing systems. dsJSON is integrated into Spark to provide SQL query processing and complex data analytics. It includes several novel components starting with the projection tree for selective parsing, and the robust partitioning techniques, as well as, the techniques for schema inference. It overcomes limitations in existing systems, and fills a gap that enables more complex analysis of large scale JSON data. Experiments on real data of up to two terabytes confirmed the scalability and efficiency of dsJSON.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF) under grants 2046236; 1838222; 1924694; 1954644; 1751392.

REFERENCES

[1] Json encoder and decoder. Available at https://docs.python.org/3/library/json.html.
[2] MongoDB. Available at https://www.mongodb.com.
[3] Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.
[4] Bestbuy developer api, 2021. Retrieved from https://bestbuyapis.github.io/api-documentation/.
[5] Jackson, 2021. Available at https://github.com/FasterXML/jackson.
[6] Jayway JsonPath, 2021. Available at https://github.com/json-path/JsonPath.
[7] RapidJSON, 2021. Available at https://rapidjson.org/.
[8] Wikipedia json dumps, 2021. Retrieved from https://dumps.wikimedia.org/wikidatawiki/latest/.
[9] Altwaijry, S. A. Y. A. H., Behm, A., Carey, V. B. Y. B. M., Cheelangi, I. C. M., Faraaz, K., Heilbron, E. G. R. G. Z., Vernica, P. P. V. T. R., Wen, J., and Westmann, T. Asterixdb: A scalable, open source bdms. *Proceedings of the VLDB Endowment 7*, 14 (2014).
[10] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data* (2015), pp. 1383–1394.
[11] Baazizi, M.-A., Berti, C., Colazzo, D., Ghelli, G., and Sartiani, C. Human-in-the-Loop Schema Inference for Massive JSON Datasets. In *EDBT 2020 - 23nd International Conference on Extending Database Technology* (Copenhagen, Denmark, Mar. 2020), OpenProceedings.org, pp. 635–638.
[12] Baazizi, M.-A., Colazzo, D., Ghelli, G., and Sartiani, C. Parametric schema inference for massive json datasets. *The VLDB Journal 28*, 4 (2019), 497–521.
[13] Barenghi, A., Crespi Reghizzi, S., Mandrioli, D., Panella, F., and Pradella, M. Parallel parsing made practical. *Science of Computer Programming 112* (2015), 195–226.
[14] Beyer, K. S., Ercegovac, V., Gemulla, R., Eltabakh, M., and Balmin, A. Jaql: A scripting language for large scale semistructured data analysis. vldb, 2011.
[15] Biswas, E. Imdb review dataset, 2021. Retrieved from https://www.kaggle.com/dsv/1836923.
[16] Bonetta, D., and Brantner, M. Fad. js: fast json data access using jit-based speculative optimizations. *Proceedings of the VLDB Endowment 10*, 12 (2017), 1778–1789.
[17] Clark, J., DeRose, S., et al. Xml path language (xpath), 1999.

[18] Čontoš, P., and Svoboda, M. Json schema inference approaches. In *Advances in Conceptual Modeling* (Cham, 2020), G. Grossmann and S. Ram, Eds., Springer International Publishing, pp. 173–183.
[19] DiScala, M., and Abadi, D. J. Automatic generation of normalized relational schemas from nested key-value data. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 295–310.
[20] Durner, D., Leis, V., and Neumann, T. *JSON Tiles: Fast Analytics on Semi-Structured Data.* Association for Computing Machinery, New York, NY, USA, 2021, p. 445–458.
[21] Eldawy, A., Hristidis, V., Ghosh, S., Saeedan, M., Sevim, A., Siddique, A., Singla, S., Sivaram, G., Vu, T., and Zhang, Y. Beast: Scalable exploratory analytics on spatio-temporal data. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management* (2021), pp. 3796–3807.
[22] Ge, C., Li, Y., Eilebrecht, E., Chandramouli, B., and Kossmann, D. Speculative distributed csv data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data* (2019), pp. 883–899.
[23] Goessner, S. JSONPath - XPath for JSON, Feb. 2007. Available at https://goessner.net/articles/JsonPath/.
[24] Hrubaru, I., Talabă, G., and Fotache, M. A basic testbed for json data processing in sql data servers. In *Proceedings of the 20th International Conference on Computer Systems and Technologies* (New York, NY, USA, 2019), CompSysTech '19, Association for Computing Machinery, p. 278–283.
[25] Jiang, L., Qiu, J., and Zhao, Z. Scalable structural index construction for json analytics. *Proc. VLDB Endow. 14*, 4 (dec 2020), 694–707.
[26] Jiang, L., Sun, X., Farooq, U., and Zhao, Z. Scalable processing of contemporary semi-structured data on commodity parallel processors-a compilation-based approach. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 79–92.
[27] Jiang, L., and Zhao, Z. Grammar-aware parallelization for scalable xpath querying. *ACM SIGPLAN Notices 52*, 8 (2017), 371–383.
[28] Jiang, L., and Zhao, Z. Jsonski: streaming semi-structured data with bit-parallel fast-forwarding. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 200–211.
[29] JSON, 2021. Available at https://www.json.org/.
[30] Documentation for the json lines text file format, 2021. Available at https://jsonlines.org.
[31] Klettke, M., Störl, U., and Scherzinger, S. Schema extraction and structural outlier detection for json-based nosql data stores. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)* (2015).
[32] Langdale, G., and Lemire, D. Parsing gigabytes of json per second. *The VLDB Journal 28*, 6 (2019), 941–960.
[33] Li, Y., Katsipoulakis, N. R., Chandramouli, B., Goldstein, J., and Kossmann, D. Mison: a fast json parser for data analytics. *Proceedings of the VLDB Endowment 10,* 10 (2017), 1118–1129.
[34] Lu, W., Chiu, K., and Pan, Y. A parallel approach to xml parsing. In *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing* (USA, 2006), GRID '06, IEEE Computer Society, p. 223–230.
[35] Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research 17*, 1 (2016), 1235–1241.
[36] Microsoft. Computer generated building footprints in all 50 us states., 2020. Retrieved from UCR-STAR https://star.cs.ucr.edu/?MSBuildings&d.
[37] Palkar, S., Abuzaid, F., Bailis, P., and Zaharia, M. Filter before you parse: Faster analytics on raw data with sparser. *Proceedings of the VLDB Endowment 11*, 11 (2018), 1576–1589.
[38] Pavlopoulou, C., Carman Jr, E. P., Westmann, T., Carey, M. J., and Tsotras, V. J. A parallel and scalable processor for json data. In *EDBT* (2018), pp. 576–587.
[39] Yip, M. Native JSON Benchmark, 2021. Available at https://github.com/miloyip/nativejson-benchmark.
[40] Zhang, Y., and Eldawy, A. Openstreetmap all map points, 2021. Retrieved from UCR-STAR https://star.cs.ucr.edu/?osm21/all_nodes&d.

Received July 2022; revised October 2022; accepted November 2022