Rumble: Data Independence for Large Messy Data Sets

Ingo Müller
ETH Zurich
ingo.mueller@inf.ethz.ch

Ghislain Fourny
ETH Zurich
ghislain.fourny@inf.ethz.ch

Stefan Irimescu*
Beekeeper AG
stefan.irimescu@beekeeper.io

Can Berker Cikis*
(unaffiliated)
canberkerwork@gmail.com

Gustavo Alonso
ETH Zurich
alonso@inf.ethz.ch

ABSTRACT

This paper introduces Rumble, a query execution engine for large, heterogeneous, and nested collections of JSON objects built on top of Apache Spark. While data sets of this type are more and more wide-spread, most existing tools are built around a tabular data model, creating an impedance mismatch for both the engine and the query interface. In contrast, Rumble uses JSONiq, a standardized language specifically designed for querying JSON documents. The key challenge in the design and implementation of Rumble is mapping the recursive structure of JSON documents and JSONiq queries onto Spark's execution primitives based on tabular data frames. Our solution is to translate a JSONiq expression into a tree of iterators that dynamically switch between local and distributed execution modes depending on the nesting level. By overcoming the impedance mismatch *in the engine*, Rumble frees the user from solving the same problem for every single query, thus increasing their productivity considerably. As we show in extensive experiments, Rumble is able to scale to large and complex data sets in the terabyte range with a similar or better performance than other engines. The results also illustrate that Codd's concept of data independence makes as much sense for heterogeneous, nested data sets as it does on highly structured tables.

PVLDB Reference Format:
Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. Rumble: Data Independence for Large Messy Data Sets. PVLDB, 14(4): 498 - 506, 2021.
doi:10.14778/3436905.3436910

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/RumbleDB/experiments-vldb21/.

1 INTRODUCTION

JSON is a wide-spread format for large data sets. Its popularity can be explained by its concise syntax, its ability to be read and written easily by both humans and machines, and its simple, yet flexible data model. Thanks to its wide support by programming languages and tools, JSON is often used to share data sets between users or systems. Furthermore, data is often first produced in JSON, for example, as messages between system components or as trace events, where its simplicity and flexibility require a very low initial development effort.

However, the simplicity of creating JSON data on the fly often leads to a certain heterogeneity of the produced data sets, which creates problems on its own. As a running example, consider the GitHub Archive, a public data set of about 1.1 TB of compressed JSON objects representing about 2.9 billion events recorded by the GitHub API between early 2011 and today. The data set combines events of different types and different versions of the API—presumably, because each event is simply archived as-is. Each individual event in the data set is already complex, consisting of nested arrays and objects with typically several dozen attributes in total. However, the variety of events makes the collection as a whole even more complex: all events *combined* have more than 1.3 k different attributes. Furthermore, about 10 % of these attributes have mixed JSON types. Analyzing this kind of data sets thus requires dealing with "messy" data, i.e., with absent values, nested objects and arrays, and heterogeneous types.

We argue that today's data analytic systems have unsatisfactory support for large, messy data sets. While many systems allow reading JSON-based data sets, they usually represent them in some flavor of data frames, which often do not work with nested arrays (for example, `pandas`) and almost always only work with homogeneous collections. Attributes with a mix of types are either dropped or kept as strings—including all nested attributes—and must be parsed manually before further analysis. For example, because there is a tiny number of integers among the objects in the `.payload.issue` path, all values at this path are stored as strings. Overall, this is the case for about a quarter of the data set, which is hence inaccessible for immediate analysis. Additionally, data-frame-based systems usually map non-existing values *and* `null` values[1] to their own representation of `NULL`, making it impossible to distinguish the two in the (admittedly rare) cases where this might be necessary. In short, the conversion to a data frame seems to be an inadequate first step for analyzing messy JSON data sets.

Furthermore, SQL is arguably not the most intuitive language when it comes to nested data, and the same is true for other languages originally designed for flat data frames or relations. While the SQL standard does define an array data type and even subqueries on individual arrays, this is done with pairs of `ARRAY(.)` and `UNNEST(.)`, which is more verbose and less readable than necessary.

*The contributions of these authors were made during their studies at ETH Zurich.

Proceedings of the VLDB Endowment, Vol. 14, No. 4 ISSN 2150-8097.
doi:10.14778/3436905.3436910

[1]These are different concepts in JSON: The attribute `foo` is non-existent in `{"bar": 42}`, but has the value `null` in `{"foo": null}`.

For example,[2] the following query selects the elements x within an array attribute arr that are less than 5 and multiplies them by 2:

```
SELECT arr, ARRAY(
    SELECT x*2 FROM UNNEST(arr) AS x WHERE x < 5)
FROM table_with_nested_attributes;
```

What is worse is that few systems implement the array functionality of the standard even partially, let alone all of it. Instead, if they support arrays at all, they do so through (often non-standard) functions such as `array_contains`, which are less expressive. For example, SQL can express "argmin-style" queries only with a self-join and none of the SQL dialects we are aware of has an equivalent of **`GROUP BY`** for arrays. Consequently, queries on messy data sets might consist of three different sub-languages: (1) SQL for the data frame holding the data set, (2) `array_*` functions for nested data, and (3) user-defined functions for otherwise unsupported tasks including parsing and cleansing the heterogeneous attributes. Current tools are thus not only based on an inadequate data model but also on a language with sub-optimal support for nested data.

In this paper, we present the design and implementation of *Rumble*, a system for analyzing large, messy data sets. It is based on the query language JSONiq [13, 30], which was designed to query collections of JSON documents, is largely based on W3C standards, and has several mature implementations. JSONiq's data model accurately represents any well-formed JSON document (in fact, any JSON document *is* automatically a valid JSONiq expression) and the language uses the same, simple and declarative constructs for all nesting levels. This makes it natural to deal with all aspects of messy data sets: absent data, nested values, and heterogeneous types. Furthermore, the fact that JSONiq is declarative opens the door for a wide range of query optimization techniques. Thanks to JSONiq, Rumble does hence not suffer from the short-comings of data-frame-based systems mentioned above but instead promises higher productivity for users working with JSON data.

The key difference of Rumble compared to previous implementations of JSONiq such as Zorba [32] and IBM WebSphere [20] is the scale of data sets it targets. To that aim, we implement Rumble on top of Apache Spark [31], such that it inherits the big data capabilities of that system, including fault tolerance, integration with several cluster managers and file systems (including HDFS, S3, Azure Blob Storage, etc) and even fully managed environments in the cloud. As a side effect, Rumble also inherits Spark's support for a large number of data formats (including Parquet, CSV, Avro, and even libSVM and ROOT, whose data models are subsets of that of JSON). In contrast, Zorba and WebSphere are designed for processing small documents and only execute queries on a single core. Rumble thus combines a high-level language that has native support for the full JSON data model with virtually unlimited scale-out capabilities, enabling users to analyze messy data sets of any scale. This also shows that the proven design principle of data independence makes just as much sense for semi-structured data as it makes for the relational domain.

2 BACKGROUND

JSON. JSON [22] (JavaScript Object Notation) is a syntax describing possibly nested values. Figure 1 shows an example. A JSON value

[2]Alternative syntax include **`LATERAL JOIN`**, for which the same arguments applies.

```
{ "type": "PushEvent",
  "commits": [{"author": "john", "sha": "e230e81"},
              {"author": "tom", "sha": "6d4f151"}],
  "repository":
      {"name": "hello-world", "fork": false},
  "created_at": "2013-08-19" }
```

Figure 1: A (simplified) example JSON object from the Github Archive data set.

is either a number, a string, one of the literals **`true`**, **`false`**, or **`null`**, an array of values, or an object, i.e., a mapping of (unique) strings to values. The syntax is extremely concise and simple, but the nestedness of the data model makes it extremely powerful.

Related are the JSON Lines [23] and ndjson [19] formats, which essentially specify a collection of JSON values to be represented with one value per line.

JSONiq. JSONiq [13, 24] is a declarative, functional, and Turing-complete query language that is largely based on W3C standards. Since it was specifically designed for querying large quantities of JSON data, many language constructs make dealing with this kind of data easy.

All JSONiq expressions manipulate ordered and potentially heterogeneous "sequences" of "items" (which are instances of the JSONiq Data Model, JDM). *Items* can be either (i) atomic values including all atomic JSON types as well as dates, binaries, etc., (ii) structured items, i.e., objects and arrays, or (iii) function items. *Sequences* are always flat, unnested automatically, and may be empty.

Many expressions work on sequences of arbitrary length, in particular, the expressions navigating documents: For example, in `$payload.commits[].author`, the object look-up operator `.` is applied to any item in `$payload` to extract their **`commits`** member and the `[]` operator is applied to any item in the result thereof to unnest their array elements as a sequence of items. Both operators return an empty sequence if the left-hand side is not an object or array, respectively. The result, all array elements of all **`commits`** members, is a single flat sequence, which is consumed by the next expression. This makes accessing nested data extremely concise. Some expressions require a sequence to be of at most or exactly one element such as the usual expressions for arithmetic, comparison, two-valued logic, string manipulation, etc.

Where it makes sense, expressions can deal with absent data: For example, the object look-up simply returns only the members of those objects that do have that member, which is the empty sequence if none of them has it, and arithmetic expressions, etc. evaluate to the empty sequence if one of the sides is the empty sequence.

For more complex queries, the FLWOR expression allows processing each item of a sequence individually. It consists of an arbitrary number of **`for`**, **`let`**, **`where`**, **`order by`**, **`group by`**, and **`return`** clauses in (almost) arbitrary order. The fact that FLWOR expressions (like any other expression) can be nested arbitrarily allows users to use the same language constructs for any nesting level of the data. For example, the following query extracts the commits from every push event that were authored by the committer who authored most of them:

```
for $e in $events          (: iterate over input :)
let $top-committer := (
  for $c in $e.commits[]  (: iterate over array :)
  group by $c.author
  stable order by count($c) descending
  return $c.author)[1]
return [$e.commits[][$$.author eq $top-committer]]
```

In the extended version of this paper [21], we present our attempt to formulate the above query in SQL, which is about five times longer than that in JSONiq.

Other expressions include literals, function calls, sequence predicates, sequence concatenation, range construction, and various "data-flow-like" expressions such as `if-then-else`, `switch`, and `try-catch` expressions. Of particular interest for JSON data sets are the expressions for object and array creation, array access, and merging of objects, as well as expressions dealing with types such as `instance-of`, `castable` and `cast`, and `typeswitch`. Finally, JSONiq comes with a rich function library including the large function library standardized by W3C.

While the ideal query language heavily depends on the taste and experience of the programmer, we believe that the language constructs summarized above for dealing with absent, nested, and heterogeneous data make JSONiq very well suited for querying large, messy data sets. In the experimental evaluation, we also show that the higher programming comfort of that language does not need to come with a large performance penalty.

3 DESIGN AND IMPLEMENTATION

3.1 Challenges and Overview

The high-level goal of Rumble is to execute arbitrary JSONiq queries on data sets of any scale that technology permits. We thus design Rumble as an application on top of Spark in order to inherit its big-data capabilities.

On a high level, Rumble follows a traditional database architecture: Queries are submitted either individually via the command line or in a shell. When a query is submitted, Rumble parses the query into an AST, which it then translates into a physical execution plan. The execution plan consists of *runtime iterators*, which each, roughly speaking, correspond to a particular JSONiq expression or FLWOR clause. Finally, the result is either shown on the screen of the user, saved to a local file, or, if the root iterator is executed in a Spark job, written to a file by Spark.

The key challenge in this design is mapping the nested and heterogeneous data model as well as the fully recursive structure of JSONiq onto the data-frame-based data model of Spark and the (mostly flat) execution primitives defined on it. In particular, since JSONiq uses the same language constructs for any nesting level, we must decide which of the nesting levels we map to Spark's distributed execution primitives.

The main novelty in the design of Rumble is the runtime iterators that can switch dynamically between different execution modes, and which encode the decision of which nesting level is distributed. In particular, iterators have one of the following execution modes: (i) *local execution*, which is done using single-threaded Java implementations (i.e., the iterator is executed independently of Spark), (ii) *RDD-based execution*, which uses Spark's RDD interface, and (iii) *DataFrame-based execution*, which uses the DataFrame interface. Typically, iterators of the outer-most expressions use the last two modes, i.e., they are implemented with higher-order Spark primitives. Their child expressions are represented as a tree of iterators using the local execution mode, which is passed as argument into these higher-order primitives.

In the remainder of this section, we first describe the runtime iterators in more detail and then present how to map JSONiq expressions onto these iterators.

3.2 Runtime Iterators

We first go into the details of the three execution modes. *Local execution* consists of Volcano-style iterators and is used for pre- or post-processing of the Spark jobs as well as for nested expressions passed as an argument to one of Spark's higher-order primitives. Simple queries may entirely run in this mode. In all of these cases, local execution usually deals with small amounts of data at the time. In addition to the usual `open()`, `hasNext()`, `next()`, and `close()` functions, our iterator interface also includes `reset()` to allow for repeated execution of nested plans. Apart from the usual semantics, the `open()` and `reset()` functions also set the dynamic context of each expression, which essentially provides access to the in-scope variables.

The other two execution modes are based on Spark. The *DataFrame-based execution* is used whenever the internal structure (or at least part of it) is known statically. This is the case for the tuple-based iterators of FLWOR clauses, where the variables bound in the tuples can be derived statically from the query and, hence, represented as columns in a DataFrame. Some expression iterators, whose output structure can be determined statically, also support this execution mode. Using the DataFrame interface in Spark is generally preferable as it results in faster execution. The *RDD-based execution* is used whenever no structure is known, which is typically the case for sequences of items. Since we represent items as instances of a polymorphic class hierarchy, which is not supported by DataFrames, we use RDDs instead.

Each execution mode uses a different interface: local execution uses the `open()`/`next()`/`close()` interface described above, while the interfaces of the other two modes mainly consist of a `getRDD()` and a `getDataFrame()` function, respectively. This allows chaining Spark primitives as long as possible and, hence to run large parts of the execution plan in a single Spark job. Each runtime iterator has a "highest" potential execution mode (where DataFrame-based execution is considered higher than RDD-based execution, which, in turn, is considered higher than local execution) and implements all interfaces up to that interface. Default implementations make it easy to support lower execution modes: by default, `getRDD()` converts a DataFrame (if it exists) into an RDD and the default implementations of the (local) `open()`/`next()`/`close()` interface materialize an RDD (if it exists) and return its items one by one. Iterator classes may override these functions with more optimal implementations. The execution modes available for a particular iterator may depend on the execution modes available for its children. Furthermore, which interface and hence execution mode is finally used also depends on the consumer, who generally chooses the highest available mode.

For example, consider the following query:

```
count( for $n in json-file("numbers.json") return $n )
```

The iterator holding the string literal "numbers.json" only supports local execution, so the iterator of the built-in function json-file() uses that mode to consume its input. Independently of the execution mode of its child, the json-file() iterator always returns an RDD. This ensures that reading data from files always happens in parallel. The iterator of the **for** clause detects that its child, the json-file() iterator, can produce an RDD, so it uses that interface to continue parallel processing. As explained in more detail in Section 3.5 below, it produces a DataFrame with a single column holding $n. The subsequent **return** clause consumes that output through the RDD interface. The **return** clause, in turn, only implements the RDD interface (since it returns a sequence of items). The $n expression nested inside the **return** clause is applied to every tuple in the input DataFrame using the local execution mode for all involved iterators. Next, the iterator of the built-in function count() detects that its child iterator (the **return** clause) can produce an RDD, so it uses Spark's count() function of RDDs to compute the result. Since count() always returns a single item, its iterator only implements the local execution mode, which is finally used by the execution driver to retrieve the query result.

In contrast, consider the following query:

```
count( for $p in 1 to 10 return $p.name )
```

Since the range iterator executing the **to** expression only supports local execution, its parent can also only offer local execution, which repeats recursively until the root iterator such that the whole query is executed locally. In particular and in contrast to the previous query, the count() iterator now uses local execution, which consumes its input one by one through next() calls incrementing a counter every time.

To summarize, Rumble can switch seamlessly between local and Spark-based execution. No iterator *needs* to know how its input is produced, whether in parallel or locally, but *can* exploit this information for higher efficiency. This allows to nest iterators arbitrarily while maintaining parallel and even the distributed DataFrame-based execution wherever possible.

The remainder of this section presents how we map JSONiq expressions to Rumble's runtime iterators. For the purpose of this presentation, we categorize the iterators as shown in Table 1 based on the implementation techniques we use. We omit the discussion of the local-only iterators as well as the local execution mode of the remaining ones as their implementations are basically translations of the pseudo-code of their specification to Java.

3.3 Sequence-transforming Expressions

A number of expressions in JSONiq transform sequences of items of any length, i.e., they work on *item**. Consider again the example:

```
$payload.commits[].author
```

The object member look-up operator . and the array unbox operator [] transform their respective input sequence by extracting the **commits** member and all array elements, respectively.

Rumble implements this type of expressions using Spark's map(), flatMap(), and filter() transformations. The function parameter given to these transformations, which is called on each item of the input RDD, consists a potentially parameterized closure that is specific to this iterator. For example, the object look-up iterator uses flatMap() and its closure is parameterized with the member name to look up ("commits" and "author" in the example above). Similarly, the unbox iterator uses flatMap() with an unparameterized closure. Note that both need to use flatMap() and not map() as they may return fewer or more than one item per input tuple. The predicate iterator used in $seq[...] is implemented with filter() and its closure is parameterized with a nested plan of runtime iterators, which is called on each item of the input RDD.

Another noteworthy example in this category is the JSONiq function annotate(), which "lifts" an RDD to a DataFrame given a schema definition by the user. It is implemented using map() and its closure attempts to convert each instance of Item to an object with the members and their types from the given schema, thus allowing for more efficient execution.

Category	Expression/Clause
local-only	(), {$k:$v}, [$seq], $$, +, -, **mod**, **div**, **idiv**, **eq**, **ne**, **gt**, **lt**, **ge**, **le**, **and**, **or**, **not**, $a\|\|$b, $f($x), $m **to** $n, **try catch**, **cast**, **castable**, **instance of**, **some** $x **in** $y **satisfies**...
sequence-producing	**json-file**, **parquet-file**, **libsvm-file**, **text-file**, **csv-file**, **avro-file**, **root- file**, **structured-json-file**, **parallelize**
sequence-transforming	$seq[...], $a[$i], $a[], $a[[]], $o.$s, $seq!..., **annotate**, **treat**
sequence-combining	$seq1,$seq2, **if** ($c) **then**... **else**..., **switch** ($x) **case**... **default**..., **typeswitch** ($x) **case**... **default**...
FLWOR	**for**, **let**, **where**, **group by**, **order by**, **count**

Table 1: Runtime iterator categorization for JSONiq expressions and clauses.

3.4 Sequence-producing Expressions

Rumble has several built-in functions that trigger distributed execution from local input: the functions reading from files of various formats as well as parallelize(). All of them take a local input (the file name and potentially some parameters or an arbitrary sequence) and produce a DataFrame or RDD.

parallelize() takes any local sequence and returns an RDD-based iterator with the same content. Semantically, it is thus the identity function; however, it allows manually triggering distributed execution from local input. An optional second argument to the function allows setting the number of Spark partitions explicitly. The implementation of this iterator is essentially a wrapper around Spark's function with the same name.

The two functions json-file() and text-file() read the content of the files matched by the given pattern either parsing each line as JSON object (i.e., reading JSON lines documents) or returning each line as string, respectively. They are RDD-based as no internal structure is known. Both are essentially wrappers around Spark's textFile(), but the iterator of json-file() additionally parses each line into instances of Rumble's polymorphic Item class.

Finally, a number of structured file formats are exposed through built-in functions that are based on DataFrames: Avro, CSV, libSVM, JSON (using Spark's schema discovery), Parquet, and ROOT. All of them have readers for Spark that produce DataFrames, which expose their "columns" or "attributes." Their implementations are again mostly wrappers around existing functionality and support for more formats can be added easily based on the same principle.

We omit the details of the sequence-combining expressions here due to space constraints. They are available in the extended version of the paper [21]. The techniques we use are very similar to the ones used for the previous two expression categories.

3.5 FLWOR Expressions

FLWOR expressions are probably the most powerful expressions in JSONiq and roughly correspond to SQL's **SELECT** statements. They consist of sequences of clauses, where each clause except **return** can occur any number of times, with the restrictions that the first clause must be either **let** or **for** and the last clause must be **return**.

The specification describes the semantics of the FLWOR expression using *streams* of *tuples*. A tuple is a binding of sequences of items to variable names and each clause passes a stream of these tuples to the next. The initial clause takes a stream of a single empty tuple as input. Finally, as explained in more detail below, the **return** clause converts its tuple stream to a sequence of items. In Rumble, we represent tuple streams as DataFrames, whose columns correspond to the variable names in the tuples.

3.5.1 For Clauses. The **for** clause is used for iteration through a sequence. It returns a tuple in the input stream for each item in the sequence where that item is bound to a new variable. If the **for** clause does not use any variable from a previous clause, it behaves like the **FROM** clause in SQL, which, if repeated, produces the Cartesian product. However, it may also recurse into local sequences such as in the example seen above:

```
for $c in $e.commits[]  (: iterate over array in $e :)
```

In the case of the first **for** clause (i.e., there were either no preceding clauses at all or only **let** clauses), the resulting tuple stream simply consists of one tuple per item of the sequence bound to the variable name introduced in the clause. In this case, the runtime iterator of the clause simply forwards the RDD or DataFrame of the sequence, lifting it from RDD to DataFrame if necessary and renaming the columns as required.

Subsequent **for** clauses are handled differently: First, the expression of the **for** clause is evaluated for each row in the input DataFrame (i.e., for each tuple in the input stream) using a Spark SQL user-defined function (UDF). The UDF is a closure parameterized with the tree of runtime iterators of the expression, takes all existing variables as input, and returns the resulting sequence of items as a `List<Item>`. In order to obtain one tuple per input tuple *and* item of those sequences, we use Spark's `EXPLODE` functionality, which is roughly equivalent to `flatMap()` on RDDs.

For example, if the current variables are `x`, `y`, and `z`, and the new variable introduced by the **for** clause is `i`, then the **for** clause is mapped to the following:

```
SELECT x, y, z, EXPLODE(UDF(x, y, z)) AS i
FROM input_stream
```

where `input_stream` refers to the input DataFrame and **UDF** is the UDF described above.

3.5.2 Let Clauses. The **let** clause is used to bind a new variable to a sequence of items. Thus, the let clause simply extends each incoming tuple to include the new variable alongside the previously existing ones. Similarly to the **for** clause, we implement the **let** clause with a UDF that executes the iterator tree of the bound expression on each input tuple:

```
SELECT x, y, z, UDF(x, y, z) AS i FROM input_stream
```

The **let** and **for** clauses allow overriding existing variables. Formally, they introduce a new variable with the same name, but since this makes the old variable inaccessible, we can drop the column corresponding to the latter from the outgoing DataFrame.

3.5.3 Where Clauses. The **where** clause filters the tuples in the input stream, passing on only those where the given expression evaluates to **true**. Similarly to the clauses discussed above, we wrap that expression into a UDF and convert the resulting value to Boolean[3] and use that UDF in a **WHERE** clause:

```
SELECT x, y, z WHERE UDF(x, y, z) FROM input_stream
```

3.5.4 Group-by Clauses. The **group by** clause groups the tuples in the input stream by a (possibly compound) key and concatenates the sequences of items of all other variables of the tuples in the same group. This is similar to SQL's **GROUP BY** clause but with the major difference that it also works without aggregation. Furthermore, the order of the groups is defined by the specification and the items in the grouping variables may be of different (atomic) types, each type producing its own group. For example, consider the following:

```
for $x in (1, 2, 2, "1", "1", "2", true, null)
group by $y := $x
return {"key": $y, "content": [$x]}
```

The result is six objects with the values `1`, `2`, `"1"`, `"2"`, **true**, and **null** in `"key"` and an array with the repeated items in `"content"`.

In short, we map this clause to a **GROUP BY** clause in Spark SQL and use `COLLECT_LIST` to combine all items of each group into one array.[4] However, this has several challenges.

First, Spark can only group by atomic, statically typed columns but the grouping variables in JSONiq are polymorphic items. We solve this problem by shredding the items in the grouping variables into three new columns. Since the grouping variables must be atomic types, there are only a few cases to consider: any number, Boolean, and **null** can be stored in a `DOUBLE` column while strings, the only remaining atomic type, can be stored in a `VARCHAR` column. Finally, we add a third column with an enum indicating the original type. This way, we can use the three new columns of each grouping variable as grouping variables in the **GROUP BY** clause. The three columns are computed by a dedicated UDF similar to what we do in the other clauses. We also add an **ORDER BY** clause on the same attributes to sort the groups as mandated by the specification.

Second, we need to recover the original items in the grouping variables. Since they cannot be grouping columns in SQL, we must use them in an aggregate function. All values of columns of the original grouping variables in one group must be the same by

[3]More precisely, we take the effective Boolean value.

[4]Interestingly, similar approaches are used for provenance tracking [16, 26, 29].

definition, so we can just pick the first one using the aggregate function `FIRST`. Finally, to keep the interface to the subsequent DataFrame simpler, we also store these as SQL arrays (which always contain one element).

Putting everything together, consider the following example query, where `x` and `y` are the grouping variables, `z` is the only non-grouping variable, and `UDF` the UDF used for shredding items:

```
SELECT ARRAY(FIRST(x)), ARRAY(FIRST(y)),
       COLLECT_LIST(z)
GROUP BY UDF(x), UDF(y)
ORDER BY UDF(x), UDF(y)
FROM input_stream
```

As an optimization, Rumble detects if a non-grouping variable is used in an aggregate function such as `count()` by one of the subsequent clauses and maps it to the corresponding aggregate function in SQL. Similarly, non-grouping variables that are not used at all are dropped.

3.5.5 Order-by Clauses. The `order by` clause returns the input tuple stream sorted by a specific variable or variables. We use a similar technique as for the sorting in the `group by` clause; however, care must be taken to fully respect the semantics of the language. The specification mandates that the sequences in the sorting variables must all be comparable; otherwise, an error must be thrown. Sequences are only comparable if (i) they contain at most one item and (ii) those items are of the same atomic type or `null`. The empty sequence is always comparable and the user can choose if it is greater or smaller than any other atomic item.

To implement this semantic, we do a first pass over the input to discover the types and throw an error if required. Then we shred the sorting attributes as described above, omitting either the column for strings or that for numbers as only one of them is used.

3.5.6 Count Clauses. The `count` clause introduces a new variable enumerating each tuple with consecutive numbers starting with 1. There is currently no functionality in Spark SQL that enumerates rows of an entire DataFrame with consecutive numbers, but practitioners have devised an algorithm [17] based on `MONOTONICALLY_INCREASING_ID()`. This function enumerates the rows *within* each partition leaving gaps in between them. The algorithm consists in enumerating the rows within each partitions using this function, then computing the partition sizes and their prefix sum, and finally broadcasting that prefix sum back into the partitions to correct for the gaps. This approach is purely based on DataFrames, runs all phases in parallel, and does not repartition the bulk of the data.

3.5.7 Return Clauses. The `return` clause ends a FLWOR expression by converting the tuple stream into the one flat sequence of items given by an expression evaluated for each tuple. We evaluate that expression with a UDF as described above, convert the DataFrame to an RDD, and unnest the sequences using `flatMap()`.

3.6 Current Limitations

With the above techniques, Rumble is able to cover the majority of JSONiq. The only two major missing features are windows in `for` clauses, which we plan to integrate soon using similar features as described above, and updates and scripting, which are on the longer-term agenda.

4 EXPERIMENTS AND RESULTS

4.1 Experimental Setup

We conduct all experiments on `m5` and `m5d` instances in AWS EC2. Unless otherwise mentioned, we process the data directly off of S3.

We use two data sets: the Github Archive [15] mentioned before as well as the *Weather* data set used by Pavlopoulou et al. [28] for evaluating VXQuery. While the latter data set is fully homogeneous and does, hence, not exercise the strengths of JSONiq, it allows for comparing our numbers with those from the original authors.

We use eight different queries: (Weather|Github)Count compute the number of records; the queries WeatherQx are taken from the original paper on VXQuery [28]; and Github(Filter|Grouping|Sorting) are simple queries that mainly consists of the corresponding operation.

We give more details of the experimental setup in the extended version of the paper [21] and a dedicated source code repository.[5]

4.2 Comparison of Distributed Engines

We first compare Rumble with four other query processing engines for large, messy data sets, i.e., cluster-based systems that can process JSON data in-situ: We use Apache AsterixDB, a system for managing and querying heterogeneous data sets; a commercial cloud-native database system that we refer to as "CloudDB;" Spark SQL running on Spark 3; and VXQuery, a former Apache project built for analyzing XML and JSON data sets that is now abandoned. For a fair comparison, we use clusters that cost about 2.5 $/h.

Usability. We express the queries for AsterixDB, CloudDB, and Spark SQL in their respective non-standard SQL dialect. The (limited) support of these dialects for nested data is enough to express the queries used in this section. However, Spark SQL does not load the objects in the `.actor` path due to a few string values at that path, so we need to use `FROM_JSON(.)` for inline JSON parsing. VXQuery uses the JSONiq extension for XQuery, which should make it easy to deal with messy data sets. However, we had to try many different reformulations of the queries to work around bugs in the query engine leading to wrong results (even for the queries proposed by the original authors), and were not able to find a correct formulation for the queries on the Github data set (including GithubCount). The SQL-based systems require the definition of external tables before any query can be run. Except for Spark, where this takes about twice as long as most of the queries, this is just a metadata operation that returns immediately. For VXQuery, the files must be copied manually into the local file system of the machines in the cluster (which we place on the local SSDs). Rumble can use the full expressiveness of JSONiq and query files on cloud storage without prior loading or setup.

Performance. Figure 2 shows the running time of the different systems on the eight queries for different subsets of the data sets. We stop all executions after 10 min. Most systems are in a similar ballpark for most queries, in particular for the Weather data set. They all incur a certain setup overhead for small data sets, which is expected for distributed query processing engines, and converge to a stable throughput for larger data sets. CloudDB is generally among the slower systems; we assume that this is because their `XSMALL`

[5]https://github.com/RumbleDB/experiments-vldb21

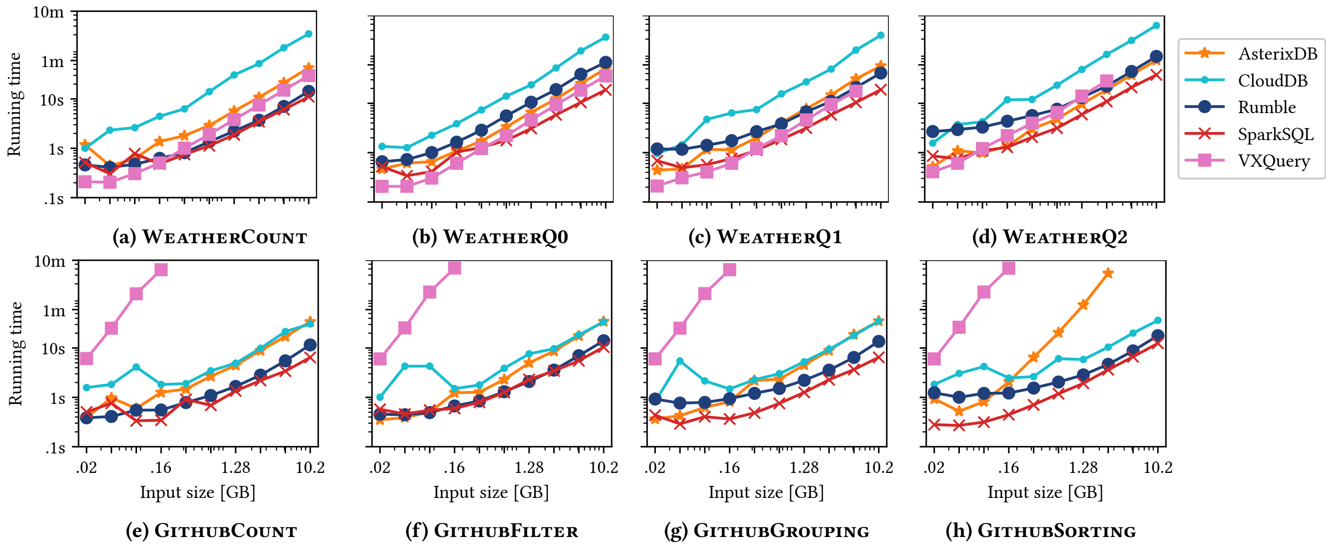


Figure 2: Performance comparison of distributed JSON processing engines.

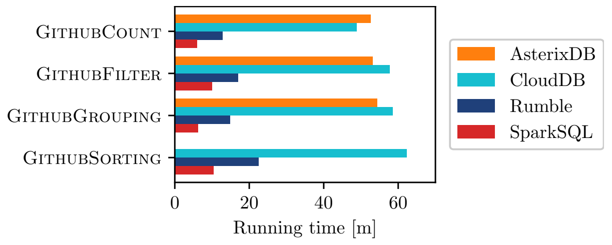


Figure 3: Distributed engines on the full Github data set.

cluster size (which is undisclosed) is smaller than the clusters of the other systems. The per-core throughput of VXQuery is in the order of 5 MB/s to 10 MB/s, which corresponds to the numbers in the original paper [28]. Note that all systems detect the self-join in WeatherQ2 and execute it with a sub-quadratic algorithm. On the Github data set, some systems show weaknesses: Curiously, VXQuery has a quadratic running time on all queries and is, hence, not able to complete queries on more than 160 MB within the time limit. (It also crashes for some configurations on the Weather data set.) Similarly, AsterixDB has a quadratic running time for the sorting query.

Rumble inherits the robustness of Spark and completes all queries without problems. Its performance is somewhat lower than that of Spark due to the interpretation overhead of its polymorphic operators and data representation. We believe that we can further tighten this gap in the future by pushing more operations and data representations down to Spark, where they would benefit from code generation and statically-typed columnar storage. Furthermore, the productivity benefits of using JSONiq, as well as its native support for heterogeneous, nested data sets make the slightly increased performance cost worthwhile.

Scalability. We also run the Github* queries on the full data set, which is about 7.6 TB large when uncompressed, and present the results in Figure 3. For this experiment, we use clusters that cost about 20 $/h, i.e., that are eight times larger than in the previous experiment. In order to avoid excessive costs, we run every query only once and stop the execution after 2 h. CloudDB and AsterixDB are not able to query the data set at this scale because a tiny number of JSON objects exceeds the maximum object size of 16 MiB and 32 MB, respectively. For CloudDB, we thus report the running time obtained after removing the problematic objects manually. This work-around also helps for AsterixDB but the system then fails with a time-out error. For reference, we plot extrapolated numbers from Figure 2 instead.[6] We do not include VXQuery here since it is not able to handle more than 160 MB in the previous experiment.

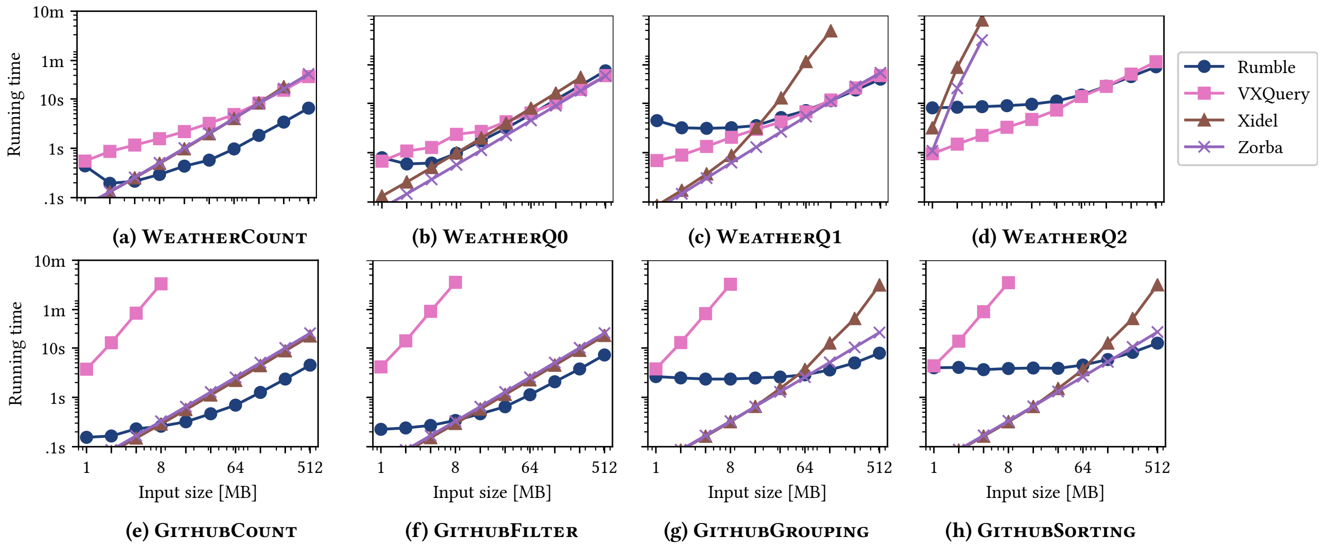
We observe that the relative performance among the systems remains as before: Rumble has a moderate performance overhead compared to Spark SQL but is significantly faster than CloudDB and AsterixDB. The experiment thus shows that Rumble can handle the full scale of the data set in terms of both size and heterogeneity while providing a high-level language tailored to messy data sets.

4.3 Comparison of JSONiq Engines

In addition to VXQuery, we now compare Rumble with two other JSONiq engines: Xidel and Zorba. Both engines are designed for small documents and hence single-threaded. In order to be able to compare the per-core performance of the engines, we configure VXQuery and Rumble to use a single thread as well. Note, however, that is not representative for typical usage on workstations and laptops, where the two engines would enjoy a speed-up roughly proportional to the number of cores. We run all experiments on `m5d.large` instances with the data loaded to the local SSD and stopped all query executions after 10 min.

Usability. All systems can read files from the local file system, though with small variations. Xidel and Zorba use the standardized file module; VXQuery uses the standardized JSONiq function

[6]This results in much more than 2 h for GithubSorting, which we, hence, omit from the plot.



(a) WeatherCount (b) WeatherQ0 (c) WeatherQ1 (d) WeatherQ2

(e) GithubCount (f) GithubFilter (g) GithubGrouping (h) GithubSorting

Figure 4: Performance comparison of JSONiq engines using a single thread.

`fn:collection`, and Rumble uses the Rumble-specific function `json-files`. Except for those of VXQuery, the remainder of the query implementations are character-by-character identical between the different systems. VXQuery has the same limitations in terms of correctness as above; the other systems behave as expected.

Performance. Figure 4 shows the results. As expected, Xidel and Zorba are considerably faster than the other two systems on small data sets, which they are designed and optimized for. In contrast, Rumble and VXQuery have a constant startup overhead for their distributed execution environment. However, the single-threaded engines struggle with larger data sets and complex queries: Xidel cannot run any query on the Weather data set of 512 MB as it runs out of main memory. Note that the data should fit comfortably into the 8 GiB of main memory. It also has a super-linear running time for the queries using sorting, grouping, or join. Zorba can handle more queries, but, like Xidel, does not seem to have implemented an equi-join as they have both quadratic running time for WeatherQ2. VXQuery handles the queries on the Weather data set well; however, it has the same quadratic running time on the Github data set as before and is hence not able to process more than 8 MB before the timeout.

Rumble can handle all queries well and, after some start-up overhead for small data sets, runs as fast as or considerably faster than the other systems. This confirms that Rumble competes with the per-core performance of state-of-the-art JSONiq engines, while at the same time being able to handle more complex queries on larger data sets. In the future, we plan to make it possible to execute queries in the local execution mode entirely in order to remove the start-up overhead for small inputs.

5 RELATED WORK

Query languages for JSON. Several languages have been proposed for querying collections of JSON documents, a substantial number of them by vendors of document stores. For example, AsterixDB [2, 3] supports the AQL language, which is maybe the most similar to JSONiq in the JSON querying landscape. Other proposals include Couchbase's N1QL [10, 11], UNQL [8], Arango's AQL [6] (homonymous to AsterixDB's), SQL++ [27], and JAQL [7]. Most of these and other proposed JSON query languages address nestedness, missing values, and `null`, but have only limited support for mixed types in the same path. We refer to the survey of Ong et al. [27] for an in-depth comparison of these languages. To the best of our knowledge, JSONiq is the only language in the survey and among those we mention that has several independent implementations.

Document stores. Document stores are related in that they provide native support for documents in JSON and similar formats [5, 9, 18]. Many of them are now mature and popular commercial products. However, document stores usually target a different use case, in which retrieving and modifying parts of individual documents are the most important operations rather than the analysis of large read-only collections.

In-situ data analysis. The paradigm employed by Rumble of query data *in-situ* has received a lot of attention in the past years. It considerably reduces the time that a scientist needs in order to start querying freshly received data sets. Notable systems include NoDB [1], VXQuery [12], and Amazon Athena [4].

Usage of Rumble. We have used Rumble for course work [25] and research on game theory [14].

6 CONCLUSION

We built Rumble, a stable and efficient JSONiq engine on top of Spark that provides data independence for heterogeneous, nested JSON data sets with no pre-loading time. Our work demonstrates that data independence for JSON processing is achievable with reasonable performance on top of large clusters. The decoupling between a logical layer with a functional, declarative language, on the one hand, and an arbitrary physical layer with a low-level query plan language, on the other hand, enables boosting data analysis productivity while riding on the coat-tails of the latest breakthroughs in terms of performance.

REFERENCES

[1] Ioannis Alagiannis, Renata Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*. DOI: 10.1145/2213836.2213864.

[2] Wail Y. Alkowaileet, Sattam Alsubaiee, Michael J. Carey, Till Westmann, and Yingyi Bu. 2016. Large-scale complex analytics on semi-structured datasets using AsterixDB and Spark. *PVLDB*, 9, 13. DOI: 10.14778/3007263.3007315.

[3] Sattam Alsubaiee et al. 2014. AsterixDB: a scalable, open source BDMS. *PVLDB*, 7, 14. DOI: 10.14778/2733085.2733096.

[4] Amazon. 2020. Athena. (2020). https://aws.amazon.com/athena/.

[5] J. Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: The Definitive Guide.* O'Reilly Media, Inc. ISBN: 978-0596155896.

[6] ArangoDB, Inc. 2020. ArangoDB. (2020). https://www.arangodb.com/.

[7] Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. 2011. Jaql: a scripting language for large scale semistructured data analysis. *PVLDB*, 4, 12. DOI: 10.14778/3402755.3402761.

[8] Peter Buneman, Mary Fernandez, and Dan Suciu. 2000. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDBJ*, 9. DOI: 10.1007/s007780050084.

[9] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage.* (2nd ed.). O'Reilly Media, Inc. ISBN: 9781449344795.

[10] Couchbase. 2018. Couchbase: NoSQL engagement database. (2018). Retrieved 10/08/2018 from http://www.couchbase.com/.

[11] Couchbase. 2018. N1QL (SQL for JSON). (2018). Retrieved 10/08/2018 from https://www.couchbase.com/products/n1ql/.

[12] Jr. E. Preston Carman, Till Westmann, Vinayak R. Borkar, Michael J. Carey, and Vassilis J. Tsotras. 2015. Apache VXQuery: a scalable XQuery implementation. In *IEEE Big Data*. DOI: 10.1109/BigData.2015.7363753.

[13] D. Florescu and G. Fourny. 2013. JSONiq: the history of a query language. *IEEE Internet Computing*, 17, 5. DOI: 10.1109/MIC.2013.97.

[14] Ghislain Fourny and Felipe Sulser. 2020. Data on the existence ratio and social utility of nash equilibria and of the perfectly transparent equilibrium. *Data in Brief*, 33. DOI: 10.1016/j.dib.2020.106623.

[15] 2020. GH archive. Retrieved 09/21/2020 from http://www.gharchive.org/.

[16] Boris Glavic and Gustavo Alonso. 2009. Perm: processing provenance and data on the same data model through query rewriting. In *ICDE*, 174–185. ISBN: 9780769535456. DOI: 10.1109/ICDE.2009.15.

[17] Evgeny Glotov. 2018. DataFrame-ified zipWithIndex. (2018). https://stackoverflow.com/a/48454000/651937.

[18] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: The Definitive Guide.* O'Reilly Media, Inc. ISBN: 978-1449358549.

[19] Thorsten Hoeger, Chris Dew, Finn Pauls, and Jim Wilson. 2016. ndjson: newline delimited JSON. (2016). Retrieved 05/01/2020 from http://www.ndjson.org/.

[20] 2017. IBM WebSphere DataPower Gateways release notes. IBM Corp. Retrieved 03/26/2020 from https://www.ibm.com/support/knowledgecenter/SS9H2Y_7.7.0/com.ibm.dp.doc/releasenotes.html.

[21] Stefan Irimescu, Can Berker Cikis, Ingo Müller, Ghislain Fourny, and Gustavo Alonso. 2019. Rumble: data independence for large messy data sets. (2019). arXiv: 1910.11582 [cs.DB].

[22] JSON. 2018. Introducing JSON. (2018). Retrieved 10/08/2018 from http://json.org/.

[23] JSON-Lines. 2018. JSON Lines. (2018). Retrieved 10/16/2018 from http://www.jsonlines.org/.

[24] JSONiq. 2018. JSONiq. (2018). Retrieved 10/08/2018 from http://jsoniq.org/.

[25] Ingo Müller, Catalina Alvarez, Mario Arduini, David Dao, Dan Graur, Susie Rao, Shuai Zhang, and Ghislain Fourny. 2020. Exercises for the Big Data course at ETH Zurich. (2020). Retrieved 12/08/2020 from https://github.com/RumbleDB/bigdata-exercises/.

[26] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. 2017. Provenance-aware query optimization. In *ICDE*. IEEE Computer Society, (May 2017), 473–484. ISBN: 9781509065431. DOI: 10.1109/ICDE.2017.104.

[27] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ unifying semi-structured query language, and an expressiveness benchmark of SQL-on-Hadoop, NoSQL and NewSQL databases. (September 2014). arXiv: 1405.3631v4.

[28] Christina Pavlopoulou, E. Preston Carman, Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. 2018. A parallel and scalable processor for json data. In *EDBT*. DOI: 10.5441/002/edbt.2018.68.

[29] Fotis Psallidas and Eugene Wu. 2018. SMOKE: fine-grained lineage at interactive speed. *PVLDB*, 11, 6, (February 2018), 719–732. ISSN: 2150-8097. DOI: 10.14778/3199517.3199522.

[30] Jonathan Robie, Ghislain Fourny, Matthias Brantner, Daniela Florescu, Till Westmann, and Markos Zaharioudakis. 2015. JSONiq: the complete reference. (2015). http://www.jsoniq.org/docs/JSONiq/html-single/index.html.

[31] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *HotCloud*.

[32] Zorba. 2018. Zorba NoSQL engine. (2018). Retrieved 10/08/2018 from http://www.zorba.io/.