Rumble: data independence when data is in a mess

Stefan Irimescu
Beekeeper AG
Zürich, Switzerland
stefan.irimescu@beekeeper.io

Can Berker Cikis
ETH Zurich
Department of Computer Science
Zürich, Switzerland
ccikis@student.ethz.ch

Ghislain Fourny
ETH Zurich
Department of Computer Science
Zürich, Switzerland
ghislain.fourny@inf.ethz.ch

Gustavo Alonso
ETH Zurich
Department of Computer Science
Zürich, Switzerland
gustavo.alonso@inf.ethz.ch

ABSTRACT
This paper introduces Rumble, an engine that executes JSONiq queries on large, heterogenous and nested collections of JSON objects, leveraging the parallel capabilities of Spark so as to provide a high degree of data independence. The design is based on two key insights: (i) how to map JSONiq expressions to Spark transformations on RDDs and (ii) how to map JSONiq FLWOR clauses to Spark SQL on DataFrames. We have developed a working implementation of these mappings showing that JSONiq can efficiently run on Spark to query billions of objects into, at least, the TB range. The JSONiq code is concise in comparison to Spark’s host languages while seamlessly supporting the nested, heterogeneous datasets that Spark SQL does not. The ability to process this kind of input, commonly found, is paramount for data cleaning and curation. The experimental analysis indicates that there is no excessive performance loss, occasionally even a gain, over Spark SQL for structured data, and a performance gain over PySpark. This demonstrates that a language such as JSONiq is a simple and viable approach to large-scale querying of denormalized, heterogeneous, arborescent datasets, in the same way as SQL can be leveraged for structured datasets. The results also illustrate that Codd’s concept of data independence makes as much sense for heterogeneous, nested datasets as it does on highly structured tables.

1 INTRODUCTION
1.1 A bit of history
When Edgar Codd abstracted away the logical concept of data tables from the physical layer [24], which is commonly known at data independence, the world of computers was dramatically different from today: the data processed back then was highly structured, and the size of a dataset would fit on a single machine. Relational database engines [12] were designed as monolithic engines running on a single machine.

The increasing amount of data available to process, as well as the ever-growing discrepancy between storage capacity, throughput and latency, have forced the database community to come up with new querying paradigms in the last two decades. Data has changed in three ways.

First (variety), data in the real world is not as structured and normalized as relational tables. Other shapes are gaining popularity: trees, graphs, cubes, and even fully unstructured data with a field of study of its own (Information Retrieval). Many products have appeared to specialize in each one of...
the data shapes: document stores [25][50][17], wide column stores [19], triple stores [53], data warehouses [23].

Second (volume), a lot of datasets (up to the Petabyte range and more) no longer fit on a single machine, meaning that a cluster is needed already only to store it. HDFS [64] is a prominent example of distributed storage system.

Third (velocity), even if the data fits on a single machine (GB or TB), disk I/O bottlenecks still require processing it in a cluster for reasonable performance.

MapReduce [27] has become very popular with its simple and general key-value abstraction, highly parallelizable and well suited to distribute I/O on clusters of machines. Spark [65] generalizes the MapReduce paradigm to a full DAG of Resilient Distributed Datasets (RDD). More recent engines take the idea further to support streaming (Flink [30], Timely Dataflows [52]) and even Machine Learning (Ray [51]).

1.2 Today’s challenge

The database landscape today is very different than what it was in the 1970s [24][12]. Today, we can store and query heterogeneous, denormalized data efficiently. But this happened at the cost of noticeably drifting away from Edgar Codd’s vision of data independence [24]. Most products require advanced programming knowledge and do not fully encapsulate layers, pushing a lot of the complexity away from the system to the user.

Platforms like MapReduce [28] or Spark [65] are runtime environments rather than programming tools. A truly data-independent system must expose data to the user via a clean data model and a functional, declarative query language, which can be executed on top of MapReduce or Spark transparently, hiding key-values or RDDs from the user.

1.3 Data independence leaks

Data independence for highly structured data is unchallenged. Most parallel computing platforms support, directly or via third-party products, an implementation of SQL (or dialect), e.g., Hive [11], Spark SQL [10]. Below are some of the most noticeable data-independence leaks of a query or program written in MapReduce, Spark, Flink, Ray, Timely, etc. We also explain why these leaks lead to actual, practical problems that affect productivity and performance.

Chains of function calls Most frameworks are based on a popular, often imperative language, e.g., Java, Scala or Python. Rather than truly offering a fully expression-based, functional language, the host language is extended with user-defined functions that are chained together. This represents a huge overhead in terms of performance and resource utilization (memory) leading to very inefficient data processing.

Query plan rather than query The chained function calls provide the look and feel of a physical query plan rather than that of a natural query language. SQL originally meant Structured English Query Language and was supposed to mirror a natural English formulation.

High number of brackets symbols The code is typically full of symbol pairs such as (), [], {}, —— and so on. This hinders readability and makes it harder to write a query as some of the focus is lost in making the code well-parenthesized.

Lambda calculus The structure of the function calls forces the user to program in a lambda calculus fashion, for example, writing a higher-order function passed as a parameter to a mapping transformation. This involves non-trivial concepts such that of closure. This makes the learning curve steep for novices with little Computer Science background. The embedding of functionality in user defined functions prevents automatic optimizations and separates tasks in ways that reduces the ability to make the execution more efficient (e.g., by merging operations, inlining steps, etc.).

Nested languages Several languages (Java, SQL, ...) may appear in the same query, nested in quoted strings or in inline syntax, the chained function calls almost constituting a language by themselves.

Mixture of different data models The same query mixes data models from the host language (e.g., Java objects), the framework (RDDs) and any nested languages (tables), introducing many impedance mismatches along the way.

2 BACKGROUND

Our contribution, Rumble, is an engine that allows writing JSONiq queries to process JSON data stored in HDFS or S3. The idea of querying data in place instead of importing into a proprietary format is enjoying growing popularity. For example, [2] provide virtual layers offering functionality from existing systems without having to move the data.

Rumble actively uses Spark internally while hiding it from the end user. In this section, we give a short introduction to JSON, Spark, and the JSONiq language, which is the existing work on which Rumble builds.

2.1 JSON

JSON [40] (JavaScript Object Notation) is a simple syntax that only contains objects, arrays, and a few atomic types: strings, numbers, booleans and null. The specification [40] is concise—a few pages.

JSON data differs from highly structured data in two ways: (i) nestedness, i.e., values can be recursively objects and arrays and (ii) heterogeneity, i.e., objects in a collection such as a JSON Lines [41] file need not have the same fields, or may associate values of different types to the same field.

JSON is the data model behind document stores such as MongoDB [50], Elasticsearch [17], Couchbase [25]. It is also the syntax of choice in which many datasets are stored and downloaded—see Figure 1 for an example.
Rumble: data independence when data is in a mess

Figure 1: An example of JSON object from the confusion dataset of the Great Language Game

SQL owes its success to the simplicity with which it manipulates tables thanks to the relational algebra. Likewise, a data-independent JSON processing engine should be based on a functional language made of expressions that manipulate collections of JSON values: objects, arrays, atomics and nothing else; a functional language similar in spirit to SQL, as powerful, but that can naturally support the heterogeneity and nestedness of JSON.

Our contribution is Rumble, a JSONiq [31][42] engine that processes large and arbitrary JSON Lines files on top of Spark, hiding away all the complex, low-level machinery of RDDs, DataFrames and transformations, and offering the user with a clean, abstract data model and a concise and expressive, highly optimizable language.

2.2 Spark

Spark [65] is a newer-generation parallel processing framework, which generalizes MapReduce to DAG data flows.

The first-class citizen of Spark is the Resilient Distributed Dataset (RDD), which is simply a flat collection of values, the concept of value being very generic—atomic values such as integers or strings, key-values, objects, etc. Input RDDs are read from layers such as HDFS and S3, or created on the fly by the client from its local memory.

RDDs are then lazily processed by transformations. Spark has a vast library of transformations of all kinds: filter, map, flatMap, mapToPair, groupByKey, sortByKey, and so on. Actions are the actual trigger of Spark computations, forcing a materialization of the output, which can be sent to the client or written back to HDFS or S3.

These primitives provide a complete toolbox, comparable to a low-level assembly language, to process data in parallel on a cluster of machines. Spark also supports DataFrames instead of RDDs for structured data. DataFrames noticeably improve performance as well as slightly improve productivity with more compact queries. However, DataFrames do not support heterogeneity. Rumble leverages DataFrames to efficiently evaluate FLWOR expressions.

2.3 JSONiq

Rumble implements JSONiq on top of Spark. JSONiq [42][31] is an expressive, declarative, functional, Turing-complete query language specifically designed for querying large quantities of JSON data. JSONiq is largely based on W3C standards.

JSONiq expressions manipulate sequences of items, which are instances of the JSONiq Data Model (JDM). An item can be (i) an atomic value, e.g., string, boolean, number, null, but also dates, binaries, etc.; (ii) an object, which maps strings to items; or (iii) an array, which is an ordered list of items. Support for such values is a pattern common to almost any modeling of arborescent data (Parquet [8], Avro [7], protocol buffers [35]...). Sequences of items are flat and do not nest, and a sequence of one item is canonically identified with that item. A sequence can be empty.

JSONiq is strongly typed. For example, (1, 2, 3, 4) matches the sequence type integer+.

A collection is typically a sequence of similar-looking object items. it is the equivalent of a table in SQL. However, unlike in the relational model, the data can be (i) heterogeneous in the sense that not all objects may have the exact same structure (ii) arborescent in the sense that objects and arrays can recursively nest. Persisting a collection is orthogonal to the language and, in the case of Rumble, data is stored in HDFS and S3.

The JSONiq syntax consists of expressions that are composed at will into an expression tree, where each expression consumes sequences of items from its child expressions, and outputs a sequence of items to its parent. An expression is evaluated in a context. Contexts contain information such as variables in scope, their types, the sequences of items with which they are bound. A parent expression may evaluate a child expression several times, with different contexts.

Expressions include literals, function calls, variables, filters, arithmetics, comparison operators, two-valued logics, dynamic object and array construction, object and array navigation, various data flow expressions such as if-then-else switch, try-catch expressions, and FLWOR expressions. JSONiq also comes with a rich function library including the large function library standardized by W3C.

The most important expression, from a database perspective, is the FLWOR expression. It is a NoSQL version of the relational algebra: project, select, join, group, order, etc. FLWOR clauses can be combined and ordered at will, making FLWOR expressions more general and fitting to NoSQL than SQL statements. For example, if we assume a large collection of objects is stored on HDFS and accessible with the function call json-file("people.json"), a JSONiq FLWOR query looks like so:

for $person in json-file("people.json")
where $person.age le 65
group by $pos := $person.position
let $count := count($person) gt 10
order by $count descending
return {
"position" : $pos,
"count" : $count
}

FLWOR expressions are usually the one scalability bottleneck of JSONiq on one machine, and at the same time the most natural expression that can be parallelized. Internally, that is, in a way that the end user does not need to be aware of, FLWOR clauses manipulate streams of tuples, which Rumble seamlessly maps to DataFrames. Likewise, Rumble seamlessly maps expression runtime iterators to RDD transformations. These mappings are explained in Section 4. We are not aware of any system capable of processing heterogeneous and nested data in a widespread and standard format at this level of performance.

3 DATA INDEPENDENCE

We now describe a number of critical issues commonly encountered when querying heterogeneous, nested datasets that are not addressed by Spark SQL. We show how to use JSONiq to address these issues, some of which have already been mentioned in Section 1.3. This leads to programs that are easier and more natural to write and read to query heterogeneous, nested datasets.

3.1 Manipulating Spark RDDs with a host language

Let us consider programs manipulating Spark RDDs written in a host language such as Java, Scala or Python. Figure 2 provides an example of the typical look-and-feel of a short query in Python. (i) Such programs are made of a chain of function calls invoking transformations, typically separated with assignments for readability; (ii) expose a query plan (map, then filter, then map, then save as file); (iii) contain a high number of various bracket symbols; (iv) use lambda constructs to pass functions as arguments; (v) the query actually contains two languages: the host language itself, and the language implied by the Spark library; (vi) the same program mixes “meta” objects such as the spark context, explicit closures, and intermediate RDD objects.

Scala and Python provide an improvement over Java that considerably simplifies the program, making it more compact, however, most of the issues regarding data independence leaks remain.

3.2 Manipulating DataFrames with Spark SQL

If the data is highly structured, DataFrames and Spark SQL provide a higher degree of data independence, as a SQL query can be passed as an argument to a method invoked on a DataFrame to produce a new DataFrame. Figure 3 shows a SparkSQL sorting query. The chain of function calls is significantly reduced, although the SQL query must be provided as a string, adding one more language to the same piece of code, together with the host language (Python), and the functional language contributed by the Spark library. The manipulated items mix Python objects, relational tables (DataFrames) and JSON values. Potential errors, such as syntax errors, will only be detected at runtime.
3.3 Manipulating structured data with JSONiq

Figure 4 shows a filtering query in JSONiq, assuming the underlying collection is highly structured and homogeneous. It can be seen that this syntax is similar to that of Spark SQL, i.e., JSONiq supports highly structured datasets as seamlessly as Spark SQL.

3.4 Manipulating heterogeneous, nested data with JSONiq

For highly structured data, Spark SQL and the like already offer a data-independent layer on top of large-scale clusters. Spark SQL supports, to some extent, nestedness with (i) a dot syntax allowing the projection of values nested inside objects and (ii) the EXPLODE() function, which normalizes a table containing array fields by creating a row for each combination of array members, and duplicating the value of other columns. For highly structured data with small levels of nestedness, Spark SQL is useful, but it becomes very verbose if the data is nested on more levels, as this translates into nested or chained Spark SQL queries.

Spark SQL reaches its limits with semi-structured or heterogeneous datasets. Figure 5 shows an example where values in a field may be of various types, or even absent. Such datasets appear when the data is unclean, in which case 95% of the values have the same type, but a few at best are absent or null, at worst have a different type. Another typical setup is when a dataset is collected over one or several decades, and the schema changes every couple of years, but without converting past data to the new schema.

Figure 6 shows the same data imported into a DataFrame: the type information is lost, as incompatible values are cast or serialized into strings. This pushes the burden of dealing with heterogeneity to the programmer, for example by guessing back the types or deserializing structures—or switching back to manipulating heterogeneous RDDs, losing the performance gains of Spark SQL.

JSONiq’s data model supports heterogeneous sequences of objects and does not require a schema. It preserves the original structure of heterogeneous and nested datasets, and its flexible syntax can convert values on the fly. Figure 7 shows an example in which the country value may have various types. In the grouping clause, a compact expression allows, on the fly at query time, to first attempt to navigate to the first value of an array; if there is none, the value of the field itself; if it is absent, a default value.

In JSONiq programs, (i) only function calls directly relevant to the semantics of the query are used, (ii) the query plan is not exposed, as this is a SELECT-FROM-WHERE like construct mirroring the English language, (iii) there are few brackets and quotes (iv) there is no need for any lambda calculus, (v) there is only one language in the query: JSONiq and (vi) there is only one data model explicitly manipulated: sequences of (potentially heterogeneous) items.

Finally, Figure 8 shows how JSONiq can handle more complex queries; there are instances of JSONiq programs with thousands of lines. JSONiq further supports updates, scripting (similar to PL/SQL) with clean snapshot semantics, as well as full-text-search extension and can thus also be used for business logics, beyond querying.

4 MAPPING THE JSONIQ DATA MODEL TO RDDS AND DATAFRAMES

4.1 Sequences of items and Spark RDDs

4.1.1 Sequences of items. As explained in Section 2, any value in JSONiq is a sequence of items, which can be directly mapped to an RDD of items.

Since Java is an object-oriented language supporting hierarchies of classes, all kinds of items can be arranged under
{  "items-ordered-on-busy-days" : [    for $order in collection("orders")      let $customer := collection("customers")        [$.cid eq $order.customer]      where $order.from eq "USA"      where every $item in $order.items        satisfies some $product        in collection("products")            satisfies $product.pid eq $item.pid      group by $date := $order.date      let $number-of-orders := count($order)      order by $number-of-orders      count $position      return {        "date": $date,        "rank": $position,        "items": [          distinct-values(            for $item in $order.items[]            for $product in collection("products")              where $product.pid eq $$.$id            return {              "name": $product.name,              "id": $product.id            }          )        ]      }  ]}

Figure 8: A more complex JSON query

an Item super class, so that an RDD of Items supports heterogeneity, i.e., sequences mixing different kinds of items: strings, numbers, booleans, arrays with all kinds of values, objects with all kinds of layouts.

4.1.2 Expressions manipulating sequences of items. Many JSONiq expressions can be parallelized using the above mapping: if all children expressions of an expression return just one sequence of items, and that sequence is physically an RDD of Items, then the semantics of that expression can often be expressed with Spark transformations and actions, while this remains hidden from the user, who only sees expressions returning sequences of items.

For example, an object lookup expression can be implemented as a flatMap transformation that applies the lookup to each object item in the RDD, filters out other items and returns the resulting RDD of items. The same goes for array navigation, predicate expressions, many functions working on sequences (tail, subsequence, ...), etc.

Another example is the count() function, which can be implemented with a count action in the RDD physically containing the parameter sequence of items. The result is not physically an RDD but a local integer – but again, the user does not see the difference: on the logical layer, it is a sequence of one item and it is irrelevant where it is physically. We will go into deeper details the general architecture and the seamless switching between Spark and local execution in Section 5.

4.2 Tuples in FLWOR expressions and Spark RDDs

A FLOWR is an expression composed of a list of clauses that ends with a return. While the entire FLWOR itself is an expression that returns a sequence of items, individual clauses (except return) return tuple streams. A tuple is a mapping between variable names and sequences of items, and this mapping contributes to the dynamic context in which nested expressions are evaluated.

Tuples can be implemented with a Tuple class carrying such a mapping – understanding that the sequences of items mapped inside a tuple are not physically RDDs, but are local materializations within the Tuple object as they are typically small.

In a first approach, we observed that, while sequences of items map elegantly to RDDs of instances of the Item class, tuple streams map elegantly to RDDs of instances of a Tuple class. Likewise, FLWOR clauses map elegantly to Spark transformations. A summary of the mappings from clauses to transformations is presented in Figure 9.

4.3 Tuples in FLWOR expressions and DataFrames

However, in a second approach, FLWOR tuples were mapped to the more modern DataFrames rather than to RDDs of Tuple objects.

Indeed, there is a fundamental difference between RDDs of Items and RDDs of Tuples: an RDD of Tuple is highly structured, because all tuples in the same tuple stream, that is, the output of any (non-return) FLWOR clause, have exactly the same in-scope variables. The values bound with the variables may be of different JSONiq sequence types, that is, what we need is DataFrames in which the type of every column (e.g., of each variable) is a List of Items.

With tuple streams represented as DataFrames, the semantics of FLWOR clauses can be implemented leveraging Spark SQL and the Catalyst optimizer on an intermediate layer,

\footnote{Note that these are not database tuples. A tuple in the context of FLWOR expressions is an assignment of variables, part of the dynamic context.}
without compromising on the support for heterogeneous sequences of items. We now explore how each FLWOR clause manipulates the physical DataFrames holding tuple streams.

### 4.4 Mapping for clauses to Spark SQL

The for clause is used for iteration through a sequence. Each incoming tuple is extended to a set of outgoing tuples, each containing all the variables from the original tuple alongside one of the values of the new variable in the current for clause. The single values are obtained from the sequence of items produced by the expression in for clause in the context of this specific incoming tuple.

Conceptually, this can be achieved with an extended projection that adds a new column to the DataFrame, corresponding to the newly bound for variable. The values in this column are obtained with a user-defined function that creates a dynamic context from the values of the other variables on the same row, evaluates the JSONiq expression of the for clause and returns the sequence containing the items to be bound in the new column. An Spark SQL EXPLODE() call then creates a new row for each such item. The resulting Spark SQL query looks like so (assuming hypothetical variables a, b and c from previous clauses, and the new variable d).

```sql
SELECT a, b, c,
    EXPLODE(EVALUATE_EXPRESSION(a, b, c)) AS d
FROM input
```

If, however, the current clause is the very first one in the FLWOR expression, then it instead creates a new DataFrame with a single column. If the underlying FLWOR expression physically supports an RDD, then for the sake of efficiency, this RDD can be mapped to a DataFrame in parallel on the cluster.

For clauses can also have positional variables. This is not supported yet as the count clause, which is implemented and offers this feature too, can be used instead. But we plan to support positional variables in the future in a way similar to count clauses.

### 4.5 Mapping let clauses to Spark SQL

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.

This is achieved in the same way as for clauses, but without the EXPLODE call.

```sql
SELECT a, b, c,
    EVALUATE_EXPRESSION(a, b, c) AS d
FROM input
```

Both the let and the for clauses support variable redeclaration. The variable will simply have the last sequence of items that was assigned to it. Prior assignments logically exist as hidden variables that are not accessible to the user and can also be dropped from the outgoing DataFrame.

If the let clause is the first clause in a FLWOR expression, we do not support the creation of a DataFrame and the execution is local. Local execution as well as dynamic switching between local and Spark execution is described in Section 5.

### 4.6 Mapping where clauses to Spark SQL

Where clauses are straightforward to describe with Spark SQL, because they are a selection on the underlying DataFrame based on the call of a user-defined function evaluating the expression in the where clause to a boolean, for each incoming tuple, like so:

```sql
SELECT a, b, c
WHERE EVALUATE_EXPRESSION(a, b, c)
FROM input
```

### 4.7 Mapping group-by clauses to Spark SQL

Group-by clauses in JSONiq are similar to the GROUP BY clause in SQL. They group incoming tuples together based on a specified key (possibly compound). A major difference in JSONiq is that all non-grouping values bound to an incoming variable are materialized as a list bound to the same (outgoing) variable. Subsequent expressions in the query may then choose to only invoke aggregation functions on this outgoing variable, or not use it at all, which straightforwardly allows optimizations since this is a functional language.

Unlike SQL, designed for highly structured and typed data, JSONiq group-by clauses also work when the keys have different types. For example, this query will not throw any error:

```sql
SELECT a, b, c
    EXPLODE(EVALUATE_EXPRESSION(a, b, c)) AS d
FROM input
```
for $i$ in parallelize(
    
    "key" : "foo", "value" : "anything"),
    "key" : 1, "value" : "anything"),
    "key" : 1, "value" : "anything"),
    "key" : "foo", "value" : "anything"),
    "key" : true, "value" : "anything"
))
group by $key := $i.key
return { "key" : $key, "count" : count($i) }

In order to benefit from the Spark optimizations, we need to first use an extended projection to create extra columns with native DataFrame types.

We create three columns for each grouping variable: The first DataFrame column is an integer that stores type information: by default 1 for an empty sequence, 2 for null, 3 for the boolean true, 4 for the boolean false, 5 for a string and 6 for a number. If however the empty sequence must compare higher than any value, then 7 is taken instead of 1. The second DataFrame column is a string that stores the value of the item bound to the sorting variable if it is a string. Otherwise, it is the empty string. The third DataFrame column is a double that stores the value of the item bound to the sorting variable if it is a number. Otherwise, it is 0.

The creation of these three columns for each variable is done in pure Java. The design of these columns is made in such a way that Spark SQL, only looking at these columns, will group the rows (tuples) in the way required by the original FLWOR order-by clause.

Once these three columns have been created for each sorting variable (say, b1, b2 and b3 for b and c1, c2 and c3 for c), it thus suffices to use the Spark SQL ORDER BY clause and project away the extra columns. We get back the original grouping keys as single items with an ARRAY_DISTINCT() call, and aggregate the non-grouping values (here a) of each group to a sequence of items with the user-defined function SEQUENCE().

\[
\text{SELECT SEQUENCE(a), ARRAY_DISTINCT(b), ARRAY_DISTINCT(c) GROUP BY b1, b2, b3, c1, c2, c3 FROM input}
\]

Rumble detects if a non-grouping variable, in consuming expressions, is aggregated as a count rather than materialized. In this, case COUNT() is invoked in Spark SQL instead of materializing the non-grouping values, for better performance. It also detects if the variable is not used in consuming expressions, in which case it does not create the column at all.

### 4.8 Mapping order-by clauses to Spark SQL

Order by clauses sort the incoming tuple stream based on the values of one or several variables. A requirement is that the sorting variables are bound to single atomic items or the empty sequence. If a sorting variable is bound to a sequence of more than one item, or to a sequence of one item that is an object or an array, then an error is returned.

Furthermore, JSONiq requires that an error be thrown in case the values associated with a variable, throughout the tuple stream, are incompatible: for example an error is thrown if there is a string and a number. The empty sequence and null, however, are comparable to any atomic value: null is smaller than any other atomic value, and the empty sequence is by default smaller than any atomic value including null (but can be changed to be greater with a keyword).

In the current implementation, a first pass is done to discover the type and throw an error in case of incompatible types. Then additional columns are created in a similar way to what is presented in Section 4.7, but only the necessary ones according to the discovered type (e.g., the number or string column can be dropped, or even both if the type is boolean).

With the additional column (for example, b1 and b2 for b, and c1 for c), the final Spark SQL query is straightforward:

\[
\text{SELECT a, b, c GROUP BY b1, b2, c1 FROM input}
\]

An alternate design would be to generate all columns as in group by, and drop the extra type check for better performance, which can be justified for large scales, but at the cost of not being fully compliant with the JSONiq specification, as some queries that should throw errors would return a result.

### 4.9 Mapping count clauses to Spark SQL

The count clause in JSONiq corresponds to a zipWithIndex transformation, which unfortunately is not available on DataFrames. We thus used the solution suggested by Evgeny Glotov on StackOverflow [34], which efficiently adds an incremental-integer DataFrame column in a parallel fashion.

### 4.10 Mapping the return clause

The return clause ends a FLWOR expression by returning the desired sequence of items, as an RDD of Items, which can be consumed by further JSONiq expressions. The return clause is implemented with a flatMap transformation, mapping each incoming tuple, i.e., row in the incoming DataFrame, to the sequence of items returned by the return expression. This results in a single, flattened RDD of items that is further consumed by parent expressions.
5 ARCHITECTURE AND IMPLEMENTATION

5.1 General architecture
Rumble follows a traditional compiler architecture inspired by the Zorba processor. The layers take a JSONiq query as input, translate it, execute it and produce results. Figure 10 presents a diagram of the architecture.

Firstly, the lexer and parser transform the query text into an Abstract Syntax Tree (AST). The lexer and parser code were automatically generated from a JSONiq grammar file using ANTLR v4. Secondly, the AST is transformed into a tree of expressions and clauses, with a class for each type of expression and clause. Thirdly (code generation), the expression and clause tree is converted to a tree of runtime iterators that encompass the semantics of each expression (returning sequences of items) and clause (returning tuple streams). Finally, runtime iterators are executed, locally or in parallel, and the results collected.

Figure 10: General Architecture Diagram

5.2 Lexer and parser
The input query is parsed with the ANTLR framework[67]. The ANTLR v4 framework uses ALL(*) [68] parsing.

The JSONiq EBNF grammar was converted to the .g4 format supported by ANTLR. From this grammar file, ANTLR generates a JSONiq lexer and parser, as well as a base class implementing the visitor pattern.

For each incoming query, the JSONiq lexer and parser are used to create the query’s Abstract Syntax Tree (AST).

5.3 Expression and clause tree generation
As in most compilers, the AST is converted to a tree of expressions and clauses using the base visitor class generated by ANTLR.

In JSONiq, the syntactic building block is an expression. FLWOR expressions are a special kind of expression further subdivided into clauses. We thus have a base abstract class that encapsulates basic functionality for any expression or clause.

Each expression has a static context including in particular in-scope variables and function declarations. In this first version, the static context mostly consists of the in-scope variables and their declared types. Keeping a distinct copies of each variable value for each expression can be costly in memory. We thus follow the common practice to chain the static contexts so that each one contains a reference to its parent context and does not duplicate any variables.

The static contexts are generated recursively by walking the tree of expressions and clauses with the visitor pattern. The static context of each expression is passed down to its children. For each variable reference expression, it is checked that the variable is present in the static context, otherwise it throws a static error. For each variable declaration, a new static context is created with new variable, and chained with the parent context.

5.4 Translation to runtime iterators
Finally, the tree of expressions and clauses is recursively converted to a tree of runtime iterators, also with a visitor pattern. Unlike traditional compilers for general-purpose imperative languages, which convert the intermediate representation into assembly instructions, it is the runtime iterators that are converted to Java Bytecode.

As explained in Section 2, expressions manipulate sequences of items and clause manipulate tuple streams. There are thus two kinds of runtime iterators, even though their internal structure is very similar.

Expression runtime iterators return items. Items are organized as a hierarchy of classes. The current version implements the core JSON item types: objects, arrays, strings, integers, doubles, decimals, booleans, strings and null, but we are working on supporting more builtin types and user-defined types.

Clause runtime iterators return tuples. Tuples are implemented as objects consisting of a map from variable names to (materialized) sequences of items.

Runtime iterators provide three APIs, between which Rumble can seamless switch back and forth: (i) local execution, to stream through the items or tuples, (ii) execution on RDDs or Items (for expression runtime iterators), and (iii) execution on DataFrames (for clause runtime iterators)\(^2\).

If the root iterator supports the RDD API, then Rumble can directly write the results back to HDFS or any other layer supported by Spark, in parallel. Otherwise (in particular for queries with a small output), it can materialize the output sequence of items to the screen or a local file. Rumble is also available on a shell, in which case the output of each query is collected (up to a configurable maximum number) and printed on the screen. The shell runs as a single Spark application, so that the executors are only set up once upon launch.

\(^2\)A previous version used to generate and consume RDDs of Tuples rather than DataFrames.
5.5 Local execution

Runtime iterators can be executed locally by obtaining the items or tuples one at a time. The architecture is pull-based, i.e., a call to retrieve the next item causes the current iterator to recursively pull one or several item from its child iterators and then process the resulting items according to its own semantics.

The local API of runtime iterators follows an established pattern: open(), hasNext(), next(), reset(), close().

A runtime iterator is always opened or reset passing a dynamic context. A dynamic context mostly binds sequences of items to variables in scope (similar to a tuple).

If the consumer invokes the runtime iterator via its local API, but it is detected that one or several child iterators support Spark and are able to return their sequence of items as an RDD, many iterators are capable of seamlessly producing their own sequence of items as an RDD with Spark transformations and actions applied to the RDDs received from the children (see Section 5.6). The RDD is then collected and materialized, and the local API simply returns the materialized items one by one. A maximum number of items to materialized can be specified and a warning is issued if the RDD has more items than this maximum. Aggregating iterators such as that of the count() function invoke a Spark count action on the child RDD to directly produce an integer item that they return.

5.6 RDD-based execution

Many expression runtime iterators also support an execution on Spark and are able to return the sequence of items as an RDD of Items. The consumer of the iterator can invoke a method isRDD(), which returns whether or the results are available as an RDD. If such is not the case, the consumer has no choice but use the local API.

If, however, the resulting sequence is available as an RDD, then another method getRDD() can be used to retrieve the sequence of items as an RDD of Items, as a more efficient alternative to the local API. The getRDD() method takes a dynamic context, dynamically applies transformations on the RDDs returned by the child iterators, according to the current iterator semantics, and finally returns the resulting RDD of Items. For example, if the expression is a object lookup (e.g., `json-file({"input.json"}).foo[].bar[$.foobar eq "a"]`) are automatically detected as being fully runnable on Spark in parallel, and the resulting output can also be written back to HDFS. None of the intermediate sequences of items is ever materialized.

5.7 Functions providing input data

In order to process FLWOR queries on top of Spark, Rumble must be able to read and write data from a storage layer such as HDFS. Two function iterators supporting the RDD API were implemented.

The json-file() function logically returns a sequence of JSON objects (internally an RDD of items) read from a text file following the json-lines standard [41]. It takes a file path and an optional number of partitions. The iterator calls Spark’s text-file() function internally using the path argument. It then maps the strings to JSON objects using Spark’s mapPartitions() and passing a mapper as argument. For best performance, we use the JSONiter [43] parser, which allows streaming to directly build the items, rather than an intermediate JSON representation.

We also provide a JSONiq wrapper for Spark’s parallelize() and with the same name. It creates an RDD of items that triggers Spark-enabled behavior in FLWOR expressions.

The consequence of the above design is that queries like

```json
json-file("input.json").foo[].bar[$.foobar eq "a"]
```

are automatically detected as being fully runnable on Spark in parallel, and the resulting output can also be written back to HDFS. None of the intermediate sequences of items is ever materialized.

5.8 DataFrame-based execution (FLWOR)

In a similar fashion to RDD-based execution for expression runtime iterators, clause runtime iterators are all able to directly return their tuple streams as DataFrames, if they detect that the input tuple stream (from the parent clause) is available as a DataFrame. The mapping from the input DataFrame to the output DataFrame is described in Section 4.4.

If the input tuple stream cannot be provided as a DataFrame, then a clause runtime iterator can use the local API of its parent clause runtime iterator to construct its output tuples, falling back to a local, pull-based execution as described in Section 5.5.
However, FLWOR expressions can also be used on small amounts of data that do not justify pushing execution down to a Spark cluster.

An initial for clause can seamlessly detect whether its expression can return the items to bind as an RDD. If so, it is capable of directly converting this input RDD of items to a DataFrame that the next clause can consume.

The return clause finally outputs the sequence of items returned by the FLWOR expression. If the previous clause provides a DataFrame, then the return clause iterator can map it to an RDD of items directly. Otherwise, it falls back to the local API to consume tuples and return items in turn.

This seamless switching between local execution and RDD/DataFrame execution is possible because JSONiq was conceived as a functional language, in order to be forward-compatible with such optimizations.

6 EXPERIMENTS AND RESULTS

This chapter presents the experiments used to analyze the performance and behavior of Rumble, as well as the obtained results. It is paramount to understand that the goal of this work is not to directly improve performance compared to existing parallel processing engines. Rather, the goal is provide an additional query language layer on top of Spark to increase productivity, with as little overhead as possible.

It is clearly expected that Rumble will be out-performed by parallel JSON document stores that pre-load data in memory such as Elasticsearch, AsterixDB, Couchbase or MongoDB, or by parallel frameworks more recent than Spark. Such systems can be extended with JSONiq support as well. With Spark as the underlying parallelization engine, using JSONiq is useful for a variety of users who do not want to incur pre-loading ETL times but directly start querying the data in place.

It is also clearly expected that Rumble will be out-performed by local engines that do not fully (or at all) support JSONiq, because they can be fine-tuned to handle very specific use cases leveraging knowledge about a specific dataset or a specific query. Rumble is a general-purpose data querying system with a Turing-complete language.

6.1 Environment, Queries and Datasets

We used several environments. The first one is a local laptop with i7-4720HQ quad-core CPU of 2.60GHz and 16GB of RAM, that allows a comparison with other locally running engines.

The second one is a 9-node cluster with entry-level configuration, each with 4 virtual cores and 16GB of RAM, namely, m5.xlarge on Amazon Elastic MapReduce. The software environment comprised HDFS 2.8.5, YARN and Apache Spark 2.4.4. A third environment comprises 10 m5.4xlarge nodes querying data stored on S3, to show that Rumble scales up seamlessly with its input.

The first dataset is the Great Language Game dataset [70]. This dataset contains over 16 million objects in the JSON Lines format, the file size being around 2.9GB. While it fits on a machine, it is large enough to cause a disk I/O. This dataset is highly structured and allows a comparison with Spark SQL. We execute three standard types of queries against this dataset: filtering, aggregating, and sorting. We also use a 20x duplication of this dataset stored on HDFS.

Finally, we used a semi-structured Reddit dataset[38] with data ranging from 2008 up to 2015, containing 54 million objects (30 GB). This dataset was replicated with a factor of 400 on S3 (up to 21.6 billion objects, 12TB).

6.2 Local execution

The first series of tests was conducted in order to compare Rumble with raw Spark (Java), Spark SQL and PySpark. All of them spread the query evaluation on the machine cores.

Figure 11 illustrates the results for the 3 types of queries presented in section 6.1, using the confusion language game dataset of around 16 million JSON objects. The complete dataset takes up around 3GB, which fits on a laptop.

Rumble competes very well with the others on the filtering query; it is faster than Spark SQL because, there, no schema inference is needed. For the grouping and sorting queries, it is at a sweet spot between Spark and Spark SQL on the one hand, and PySpark on the other hand.

All in all, the productivity benefits of using JSONiq, as well as its native support for heterogeneous, nested datasets make the slight incurred performance cost worthwhile, and not higher than that of using the popular and productive Python language.
6.3 Comparison with other JSONiq engines

Figure 12: Comparison of JSONiq queries between Rumble (blue), Zorba (orange) and Xidel (grey) for (i) a filter query, (ii) a grouping query and (ii) a sorting query. The measurements were capped at 600 seconds.

6.4 Cluster execution

We ran measurements on the same queries, comparing against Spark in Java, Spark SQL, Pyspark and Rumble, on our cluster of 9 nodes. The runtimes are, like before, end-to-end and the results closely relate to local measurements. The dataset was duplicated 20 times, to 320M objects weighing 58GB. Figure 13 shows that JSONiq performs best for filtering queries, runs as fast as raw Spark on a sorting query, and is twice slower than raw Spark or Spark SQL on grouping. Rumble is faster to Pyspark on all queries, demonstrating a sweet spot between productivity and performance.
6.5 Speedup analysis

Speedup analysis experiments were also executed in order to observe the behavior of Rumble in more detail. Figure 14 shows the runtime for a highly filtering query on the 30GB Reddit dataset for different numbers of executors between 1 and 32, on the same 9-node cluster. It also shows the aggregated runtime over the cluster, which slightly goes up as computations are spread more over the cluster, ending at no more than a factor of 2. Measurements are averaged over 5 tries. We observed at times a few artefacts due to YARN resource allocation that are independent of Rumble.

6.6 Processing large datasets

In order to see how Rumble behaves in Big-Data scenarios, the final set of tests was carried out with a duplicated Reddit dataset from 2007 to 2015 of 54 million documents, replicated up to 400 times. This means that the largest collection had 21.6 billion objects (12 TB). The cluster was scaled up and out to 1+9 m5.4xlarge EC2 machines, the maximum default quota. The result of a filtering query for different sizes (subsets) are shown on Figure 15. We can see that the curve is very linear, and the number of machines is relatively small, meaning that the limits of Rumble are not reached and larger collections are likely to be supported as well. This shows that Rumble successfully rides on the coat-tails of Spark in terms of scalability. It is planned in the future to find out the scale at which Rumble starts showing weakness.

7 RELATED WORK

There have been efforts to address some of the same shortcomings we address in this paper but these efforts are either undocumented, under documented, have stopped, or are not accessible for us to compare with our system.

7.1 Apache VXQuery

Apache VXQuery [29] is a parallel and scalable XML query processor implemented in Java. VXQuery uses the Hyracks parallel execution engine, which predates Spark, and Algebrics, a compiler toolbox also used by AsterixDB.

VXQuery recently introduced JSONiq support [56]. VXQuery, like Rumble, focuses on use cases in which the JSON data is externally stored and not preloaded, for more flexibility. In this respect, VXQuery and Rumble both make a point that JSONiq can be used as a standardized JSON processing language on top of various platforms.

VXQuery supports both JSON and XML while Rumble is more lightweight and supports only JSON. Rumble is a runtime-iterator-based engine that flexibly and seamlessly switches between local vs. Spark execution, while VXQuery is based on Hyracks (physical plan) and Algebricks (logical plan), and produces a query plan directly from the AST, modified with rewrite rules.

16 cores and 128 GB per machine
As we write these lines, JSONiq support in VXQuery is not publicly released, and the master was not usable in its current state in spite of our support requests. A performance comparison was thus not possible.

### 7.2 JSON querying languages

JSONiq was introduced in 2010. It has a full and mature specification available online, is supported by several independent engines and is made of 90% of material – including FLWOR expressions – standardized by the W3C.

A few other languages have appeared in the last decade. Most proposed JSON query languages address nestedness and partial heterogeneity, missing or null values, but not incompatible types. They support recursive structures with objects and arrays, but large collections queried with SELECT-FROM-WHERE statements are often limited to large sequences of objects in which each queried field must have either a specified type, or be null or missing.

Some JSON document stores provide a data independent layer with a more complete language. AsterixDB [4] [3] supports the AQL language, which is the most similar to JSONiq in the JSON querying landscape. Other proposals include Couchbase’s NiQL [25][26], UNQL [16], Arango [9]’s AQL (homonymous to AsterixDB’s), SQL++ [54], JAQL [46], Drill [6], Postgres-XL [59].

There are also variants of JSONiq adapted to languages such as PythonQL [60], similar to C#’s LINQ [49]. MRQL was an alternate proposal running on MapReduce, Spark, Flink and Hama, also similar in spirit to JSONiq, however it was stopped in 2017. A logical model for querying JSON was contributed in [15].

### 7.3 Document stores

Document stores provide native arborescent storage and querying, typically supporting JSON [40] or closely related data models [21][22][39]. Many of them are now mature and popular commercial products. Some, like Elasticsearch [17] and MongoDB [50] have their own low-level JSON-based query languages, optimized for simple and fast operations such as projection and selection, as well as more evolved features following paradigms such as MapReduce.

The performance of document stores come at the cost of having to pre-loaded (ETL) data. This requires large efforts both in terms of transfer time, impedance mismatch solving and often manual work.

### 7.4 Querying data in place

Rumble allows its users to directly query the data in place on S3 or HDFS with no need to move it to a data store. This considerably reduces the time that a scientist needs in order to start querying freshly received datasets. Querying raw data directly, rather than loading it into a product’s optimized format is also an active avenue of research [2] [29] [5] [13]. CleanM [33] is a query language specifically designed for data cleaning at large scales.

### 7.5 Parallel processing engines

MapReduce [28] is a simple parallel processing framework that processes a collection of key-value pairs in two phases. First, the input collection gets mapped to an intermediate collection. Then, these intermediate key-values get shuffled over the network in such a way that the same keys are on the same machine. Then, the final reducing step generates an output collection.

Flink [30] is a low-latency, high-throughput, in-memory, scalable streaming engine that supports stateful computations and unbounded streams. Likewise, Timely dataflow [52] is a powerful programming model and distributed engine that achieves expressiveness and high performance for processing streams and graphs. It was implemented in Rust [62]. Ray [51] is a high-throughput, low-latency computing engine specifically targeting learning reinforcement use cases like autonomous cars and unmanned drones, and thus mainly supports machine learning features. Ray also supports an imperative taste, and it can also leverage Apache Spark.

Other engines provide additional features on top of parallel hardware, such as top-K query processing [63], extracting structure [32], analytics [14], imperative Java programs [1].

### 7.6 Other data models

Besides tables and trees, models include cubes and graphs.

Apache Hive [18] is a data warehouse project that aims to bring a unified, SQL-like interface in order to query data stored on HDFS (HiveQL)[11]. The way it works is that it transparently converts SQL queries to MapReduce jobs or even chains of Spark transformations and actions. Shark [69] improves on its performance. AQP++ provides interactive analytics on datacubes [57].

GraphX is a Spark API designed to process graphs and “graph-parallel computations” [61]. It introduces a new abstraction, similar to the basic Spark RDD, called “Resilient Distributed Graph” which is designed to help with graph creation, loading and computations.

Supporting several models with the same engine is also an avenue of research enjoying increasing popularity, including the integration of JSON into relational databases [47] [45] [66] [37] [44] [48] [20] [58] [55].

### 8 CONCLUSION AND FUTURE WORK

We built Rumble, a stable and efficient JSONiq engine on top of Spark to provide data independence for heterogeneous,
nested JSON datasets with no pre-loading time. Rumble is publicly available as a stable open-source release, under Apache 2.0.

Most features of JSONiq are supported, and comprehensive support is an active, ongoing effort with no major issues expected. Some major features yet to be added include user-defined functions and modules, additional types, try-catch expressions, schema validation, and we continue to add more standardized functions and to push-down more expressions to Spark. Further work includes the implementation of JSONiq on top of other engines such as Timely [52], Flink [30] and Ray [51], testing with other storage layers, as well as adding support for FLWOR window clauses for platforms that support streaming.

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] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In SIGMOD.

[2] Ioannis Alagiannis, Renato Borovica-Gajic, Miguel Branco, Stratos Idreos, and Anastasia Adlamaki. 2015. NoDB: Efficient Query Execution on Raw Data Files. Communications of the ACM. Research Highlights (2015).

[3] Wail Y. Alkowaileet, Sattam Alsabiae, Michael J. Carey, Till Westmann, and Yingyi Bu. 2016. Large-scale Complex Analytics on Semi-structured Datasets Using asterixDB and Spark. Proc. VLDB Endow. 9, 13 (Sept. 2016), 1585–1588. https://doi.org/10.14778/3007263.3007315

[4] Sattam Alsabiae, Yasser Alkowail, Hotham Altwaijry, Alexander Behm, Vinayak Borkar, Yingyi Bu, Michael Carey, Cici Cetindil, Madhusudan Cheelangi, Khuuram Faraz, Eugenia Gabriellova, Raman Grover, Zachary Heilbron, Young-Seok Kim, Chen Li, Guangqiang Li, Ji Mahn Ok, Nicola Onose, Pouria Pirzadeh, Vassilis Tsotras, Rares Vernica, Jian Wen, and Till Westermann. 2014. AsterixDB: A Scalable, Open Source BDM. Proc. VLDB Endow. 7, 14 (Oct. 2014), 1905–1916. https://doi.org/10.14778/2733085.2733096

[5] Amazon. [n. d.]. Athena. https://aws.amazon.com/athena/.

[6] Apache. [n. d.]. Drill. http://drill.apache.org/.

[7] Apache. 2018. Avro. https://avro.apache.org/.

[8] Apache. 2018. Parquet. https://parquet.apache.org/.

[9] ArangoDB. [n. d.]. ArangoDB. https://www.arangodb.com/

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davide Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Aftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD ’15). ACM, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[11] Namit Jain Zheng Shao Prasad Chakka Suresh Anthony Hao Liu Pei Wyckoff Ashish Thusoo, Joydeep Sen Sarma and Raghotham Murthy. 2010. Hive - A Warehousing Solution Over a Map-Reduce Framework. (2010).

[12] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. R. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Melih, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. 1976. System R: Relational Approach to Database Management. ACM Trans. Database Syst. 1, 2 (June 1976), 97–137. https://doi.org/10.1145/320455.320457

[13] Amazon Athena. [n. d.]. https://docs.aws.amazon.com/athena/latest/ug/json.html.

[14] Mike Barnett Robert DeLine Danyel Fisher Badrish Chandramouli, Jonathan Goldstein, John C Platt, James F Terwilliger, and John Wernsing. 2014. Trill: A high-performance incremental query processor for diverse analytics. In VLDB.

[15] Pierre Bourhis, Juan L. Reutter, Fernando Suarez, and Domagoj Vrgoč. 2017. JSON: Data Model, Query Languages and Schema Specification. In Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS ’17). ACM, New York, NY, USA, 123–135. https://doi.org/10.1145/3034786.3056120

[16] Peter Buneman, Mary Fernandez, and Dan Suciu. 2000. UnQL: A Query Language and Algebra for Semi-structured Data Based on Structural Recursion. The VLDB Journal (2000).

[17] Elasticsearch BV. 2018. Elasticsearch. http://www.elasticsearch.co/. [Online; accessed 8-October-2018].

[18] Jesús Camacho-Rodríguez, Ashutosh Chauhan, Alan Gates, Eugene Kofman, Owen O’Malley, Vineet Garg, Zoltan Haindrich, Sergey Shelukhin, Prasanth Jayachandran, Siddharth Seth, Deepak Jaiswal, Slim Bouguerra, Nishant Bangarwa, Sanka Hariappan, Anishbek Agarwal, Jason Dere, Daniel Dai, Thejas Nair, Nita Dembla, Gopal Vayajravahyan, and Günther Hageltrner. 2019. Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. SIGMOD (2019).

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 205–218.

[20] Craig Chasseur, Yinan Li, and Jignesh M Patel. 2013. Enabling JSON Document Stores in Relational Systems. In WebDB, Vol. 13. 14–15.

[21] Kristina Chodorow. 2013. MongoDB: The Definitive Guide: Powerful and Scalable Data Storage. O’REILLY.

[22] Zachary Tong Clinton Gormley. 2015. Elasticsearch: The Definitive Guide. O’REILLY.

[23] E.F. Codd, S.B. Codd, and C.T. Salley. 1993. Providing OLAP (On-line Analytical Processing) to User-analysts: An IT Mandate. Codd & Associates. https://books.google.ch/books?id=pt0lGwAACAAJ

[24] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[25] couchbase. 2018. Couchbase: NoSQL Engagement Database. http: //www.couchbase.com/. [Online; accessed 8-October-2018].

[26] Couchbase. 2018. N1QL (SQL for JSON). https://www.couchbase.com/products/n1ql/. [Online; accessed 8-October-2018].

[27] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In OSDI’04: Sixth Symposium on Operating System Design and Implementation. San Francisco, CA, 137–150.

[28] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. OSDI (2004).

[29] JR. E. Preston Carman, Till Westermann, Vinayak R. Borkar, Michael J. Carey, and Vassilis J. Tsotras. [n. d.]. Apache VXQuery: A Scalable
XQuery Implementation. ([n. d.])

[30] flink. 2018. Apache Flink - Stateful Computation over Data Streams. http://flink.apache.org/. [Online; accessed 8-October-2018].

[31] D. Florescu and G. Fourny. 2013. JSONiq: The History of a Query Language. IEEE Internet Computing 17, 5 (Sept 2013), 86–90. https://doi.org/10.1109/MIC.2013.97

[32] Yihan Gao, Silu Huang, and Aditya Parameswaran. 2018. Navigating the Data Lake with Datamaram: Automatically Extracting Structure from Log Datasets. In SIGMOD.

[33] Manos Giannakopoulou, Benjamin Gaidioz Karpathiotakis, and Aliyamaki Anastasia. 2017. CleanM: An Optimizable Query Language for Unified Scale-Out Data Cleaning. VLDB (2017).

[34] Evgeny Glotov. [n. d.]. DataFrame-ified zipWithIndex. https://stackoverflow.com/questions/30304810/dataframe-ified-zipwithindex. [Online; accessed 8-October-2018].

[35] Google. 2018. Protocol buffers. https://developers.google.com/protocol-buffers/. [Online; accessed 17-October-2018].

[36] Patrick Wendell Holden Karau, Andy Konwinski and Matei Zaharia. Google. 2018. Protocol buffers. https://developers.google.com/protocol-buffers/. [Online; accessed 17-October-2018].

[37] Reddit User Stuck in the Matrix. 2018. Reddit dataset. https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/. [Online; accessed 8-October-2018].

[38] Noah Slater J. Chris Anderson, Jan Lehnardt. 2014. CouchDB: The Definitive Guide. O’REILLY.

[39] [49] Microsoft. 2018. LINQ – Language Integrated Query. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/. [Online; accessed 17-October-2018].

[50] MongoDB. 2018. MongoDB. http://www.mongodb.com/. [Online; accessed 8-October-2018].

[51] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. CoRR abs/1712.05889 (2017).

[52] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, Iterative Data Processing with Timely Dataflow. Commun. ACM 59, 10 (Sept 2016), 75–83. https://doi.org/10.1145/2983551

[53] Neo4j. 2018. The Neo4j Graph Platform. http://www.neo4j.com/. [Online; accessed 8-October-2018].

[54] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. [n. d.]. The SQL++ Query Language: Configurable, Unifying and Semi-structured. ([n. d.]).

[55] Steven Ortiz, Caner Enbatan, Maksim Podkorytov, Dylan Soderman, and Michael Gubanov. 2017. Hybrid. json: High-velocity parallel in-memory polystore JSON ingest. In 2017 IEEE International Conference on Big Data (Big Data), IEEE, 4807–4809.

[56] 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.

[57] Jinglin Peng, Dongxiang Zhang, Jianman Wang, and Jian Pei. 2018. AQPl++: Connecting Approximate Query Processing With Aggregate Precomputation for Interactive Analytics. In SIGMOD.

[58] Dušan Petković. 2017. JSON integration in relational database systems. Int J Comput Appl 168, 5 (2017), 14–19.

[59] Postgres-XL. [n. d.]. Postgres-XL. http://postgres-xl.org/

[60] pythonql. 2018. PythonQL. https://pypi.org/project/pythonql/. [Online; accessed 17-October-2018].

[61] Robert W. Quong T. J. Parr. 1995. ANTLR: A Predicated-LL(k) Parser. R. W. Quong T. J. Parr. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. (2014).

[62] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and Rich Analytics at Scale. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD ’13). ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/2463676.2465288

[63] Lars Yencken. 2014. The Great Language Game. Blog Post.

[64] D. Florescu and G. Fourny. 2013. JSONiq: The History of a Query Language. IEEE Internet Computing 17, 5 (Sept 2013), 86–90. https://doi.org/10.1109/MIC.2013.97

[65] MongoDB. 2018. MongoDB. http://www.mongodb.com/. [Online; accessed 8-October-2018].

[66] Michael J. Franklin Ion Stoica, and Michael Gubanov. 2017. Hybrid. json: High-velocity parallel in-memory polystore JSON ingest. In 2017 IEEE International Conference on Big Data (Big Data), IEEE, 4807–4809.

[67] Christine Pavlopoulou, E. Preston Carman, Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. 2018. A Parallel and Scalable Processor for JSON Data. In EDBT.

[68] Rust. 2018. The Rust Programming Language. https://www.rust-lang.org/. [Online; accessed 22-October-2018].

[69] Anil Shanbhag, Holger Pirk, and Samuel Madden. 2018. Efficient Top-K Query Processing on Massively Parallel Hardware. SIGMOD (2018).

[70] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). 1–10. https://doi.org/10.1109/MSST.2010.5496972

[71] spark. 2018. Apache Spark - Unified Analytics Engine. http://spark.apache.org/. [Online; accessed 8-October-2018].

[72] Michael Stonebraker. 2013. The case for polystores. http://wp.sigmod.org/?p=1629.

[73] R. W. QUONG T. J. PARR. 1995. ANTLR-A Predicated-LL(k) Parser Generator. (1995).

[74] R. W. QUONG T. J. PARR. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. (2014).

[75] R. W. QUONG T. J. PARR. 1995. ANTLR-A Predicated-LL(k) Parser Generator. (1995).

[76] R. W. QUONG T. J. PARR. 2014. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. (2014).

[77] R. W. QUONG T. J. PARR. 1995. ANTLR-A Predicated-LL(k) Parser Generator. (1995).