Master Thesis

JSONiq on Spark

Author(s):
Irimescu, Stefan

Publication Date:
2018

Permanent Link:
https://doi.org/10.3929/ethz-b-000272416

Rights / License:
In Copyright - Non-Commercial Use Permitted
JSONiq on Spark

Master Thesis
Stefan Irimescu
August 27, 2017

Advisors: Prof. Dr. Gustavo Alonso, Dr. Ghislain Fourny
Department of Computer Science, Systems Group, ETH Zürich
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Background and Related Work</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>NoSQL and Big Data</td>
<td>3</td>
</tr>
<tr>
<td>2.2</td>
<td>MapReduce, Hadoop and YARN</td>
<td>4</td>
</tr>
<tr>
<td>2.3</td>
<td>Spark</td>
<td>5</td>
</tr>
<tr>
<td>2.3.1</td>
<td>RDDs</td>
<td>5</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Creating RDDs</td>
<td>6</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Transformations</td>
<td>6</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Actions</td>
<td>7</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Running on a Cluster</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>JSONiq</td>
<td>9</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Core Language</td>
<td>9</td>
</tr>
<tr>
<td>2.4.2</td>
<td>FLOWR Expressions and Tuple Streams</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Related Work</td>
<td>16</td>
</tr>
<tr>
<td>2.5.1</td>
<td>Apache VXQuery</td>
<td>16</td>
</tr>
<tr>
<td>2.5.2</td>
<td>Document Stores</td>
<td>17</td>
</tr>
<tr>
<td>2.5.3</td>
<td>Other JSON Processing Languages</td>
<td>17</td>
</tr>
<tr>
<td>2.5.4</td>
<td>Data Models for Parallel Frameworks</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Theoretical Aspects of Parallel JSONiq Queries</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>Tuple Streams to RDDs</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>FLWOR Clauses to Spark Transformation</td>
<td>19</td>
</tr>
<tr>
<td>3.2.1</td>
<td>For Mapping</td>
<td>20</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Let Mapping</td>
<td>21</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Where Mapping</td>
<td>22</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Order By Mapping</td>
<td>22</td>
</tr>
<tr>
<td>3.2.5</td>
<td>Group By Mapping</td>
<td>23</td>
</tr>
</tbody>
</table>
3.2.6 Return Mapping ........................................ 24
3.2.7 Summary and Example ................................. 25

4 Project Implementation ...................................... 27
4.1 General Architecture ...................................... 27
4.2 Compiler .................................................. 28
  4.2.1 Lexer and Parser ...................................... 28
  4.2.2 Expression Tree Generation ......................... 28
  4.2.3 Static Context ...................................... 29
  4.2.4 Item Hierarchy ................................... 32
4.3 Runtime Iterators ......................................... 33
  4.3.1 Translator and Dynamic Context ................... 34
4.4 Spark-Enabled Iterators .................................. 35
  4.4.1 Spark-Enabled Function Iterators .................. 36
4.5 FLWOR Iterators ........................................... 39
  4.5.1 Local FLWOR Implementation ....................... 39
  4.5.2 FLWOR Tuples and Closures ......................... 40
  4.5.3 For and Return Clauses Implementation ............ 41
  4.5.4 Where Clause Implementation ....................... 43
  4.5.5 Let Clause Implementation .......................... 43
  4.5.6 Order By Clause Implementation .................... 44
  4.5.7 Order-by Clause .................................... 45
  4.5.8 Group By Clause Implementation .................... 46
4.6 Additional Components .................................... 47
  4.6.1 Testing Framework .................................. 47
  4.6.2 Shell and Command Line Execution Options ........ 47
4.7 Build Tools, Dependencies, Licenses and Execution .... 49
  4.7.1 Build ................................................ 49
  4.7.2 Dependencies/Licenses .............................. 49

5 Experiments and Results ..................................... 51
5.1 Environment, Queries and Datasets ....................... 51
5.2 Serialization Performance ................................ 52
5.3 Break-Even Point Analysis ................................ 53
5.4 Comparison with Elasticsearch ............................ 54
5.5 Speedup Analysis ........................................ 57
5.6 Processing Large Datasets ................................ 57

6 Conclusions and Future Work ............................... 59

A Appendix ..................................................... 61
  A.1 Implemented Features .................................... 61

Bibliography .................................................... 63
Chapter 1

Introduction

New querying paradigms have been created by the database community because of the increasing demands of big data and the discrepancy between storage capacity, throughput and latency. MapReduce has become very popular because of its simple, general abstraction. Spark improved upon the foundation of MapReduce while making it more general, shifting from a strictly two-phase model to a full DAG of phases. These tools bring us closer to the goal of having a truly data-independent system which exposes the data to the user via a functional, declarative query language and a clean data model.

The purpose of this master thesis is to implement a simple JSONiq engine that can process large JSON files on top of Spark. The paradigm for JSONiq FLWOR expressions is mapped to that of Spark, tuple streams are mapped to RDDs and FLWOR clauses to Spark transformations and actions. The engine can forward the execution of FLWOR clauses to Spark without exposing this to the user.

This project is intended to be a first step toward distributed, Spark-enabled document stores.
Chapter 2

Background and Related Work

This chapter is meant to give an introduction to the context of this project including the current state of Big Data related technologies, NoSQL, JSONiq and Spark.

2.1 NoSQL and Big Data

The term “big data” may be a buzzword but in general, it can be used to describe any data set that is too large to be handled by traditional data processing technologies. Big data can be described by a multitude of characteristics, the most important being [7]:

1. Volume - the ever growing amount of data that gets collected comes with increasing demands on storage and processing technologies. Looking at the scientific community, CERN produces around 30PB of data every year while the Sloan Digital Sky Survey can gather up to 200GB of data every night, constructing a detailed 3D map of the universe. But it is not just the scientific community that generates massive amounts of data. Today, data is an automatically generated byproduct of human activities and the fact that data carries value is a strong incentive to ever increasing data gathering.

2. Variety - data is now collected from various sources ranging from scientific experiments to sensor data and content usage logs on the web. Given this large spectrum of sources, data now comes in a variety of shapes. While plain text and tables are still significant, new shapes such as graphs, trees and cubes are being adopted and choosing the right data shape for a given scenario is crucial.

3. Velocity - the ability to effectively handle and process data is described by 3 factors: capacity, throughput and latency and in the past 50 years, there has been an ever increasing discrepancy between these factors
While storage capacity has increased by a factor in the 9 digits, there has only been an improvement of 8x in latency.

While traditional relational database management systems or RDBMS have been the standard in the past 40 years, the increasing demands of big-data-related applications now ensure the decline of this database technology. The fact that RDBMS maintain a highly structured form of the data has clear advantages when the total amount of information is small but at the same time it is a significant impediment when scalability is needed. Increasing the data amount beyond certain limits results in very significant performance penalties when using traditional relational models.

NoSQL (sometimes called “Not Only SQL”) refers to different database frameworks that mean to provide high performance and scalability for a massive scale. In general, this is done by trading the highly structured data form present in RDBMS for speed and efficiency. There are many different database technologies that can be classified as NoSQL but all of them are designed with the concept of distributed databases in mind. Being able to query and process unstructured or semi-structured data stored on multiple nodes is essential for big-data-oriented applications. This ability to scale horizontally means that when the amount of data grows, more hardware resources can be added and the performance remains the same, with no additional slowdowns. Some examples of NoSQL technologies include key-value stores, column-oriented stores and document stores.

2.2 MapReduce, Hadoop and YARN

One of the most popular frameworks for data processing on a cluster is MapReduce[5]. It provides a general, easy to use abstraction by splitting the process in three steps:

- **Map**: Workers call a map() function on local data and output results to some temporary location.

- **Shuffle**: Workers redistribute data based on the output keys produced in the previous step. The step is complete when all data belonging to a single key is located on one worker.

- **Reduce**: Workers, each one with an unique key, process the data in parallel.

The simplicity of MapReduce alongside with its ability to be easily adapted over a distributed storage layer such as Apache HDFS[12] have ensured its success but limitations of the framework are drawing increasing criticism. The shuffling phase can be very problematic when targeting high performance in real-life scenarios and also, there is the general inflexibility of the MapReduce topology, which limits usability. More problems arise when
running jobs on top of an HDFS cluster. Assigning the additional role of MapReduce job tracker to the name node (central master node) on a HDFS cluster basically transforms the name node into a jack-of-all-trades and can bottleneck the entire system. Naturally, this approach does not scale well beyond a certain number of nodes.

More recent releases of Apache Hadoop have improved the situation by introducing YARN[18] (acronym for Yet Another Resource Negotiator), a new cluster resource manager. YARN attempts to solve the scalability issues by redistributing the work that was entirely allocated to the central master in the cluster. When the client submits a job, the central master which acts as a scheduler, allocates an “application master” somewhere on the cluster. Then, this newly created application master is responsible for resource requests and job monitoring, thus removing this responsibility from the name node. This increases scalability from tens of thousands of nodes up to hundreds of thousands of nodes.

YARN also offers more flexibility than the traditional MapReduce topology, which is very strict, basically allowing processes made up of any DAG.

2.3 Spark

Spark is a cluster computing framework which improves on most aspects of previous technologies, primarily focusing on speed and flexibility[13]. It’s fully compatible with YARN and other cluster managers. Spark moves away from the specific structure of MapReduce and enables computations of any DAG by introducing a new concept, namely, the RDD.

2.3.1 RDDs

The name RDD stands for Resilient Distributed Dataset and is the core abstraction for data processing in Spark. An RDD is an immutable collection of elements that Spark splits into multiple partitions under the hood and those separate partitions may be computed on different cluster nodes. An RDD life-cycle consists of 3 stages:

- Creation - an RDD is created either by loading an external dataset from various sources, or by generating a collection of items programmatically.

- Transformation - new RDDs can be created from previously existing ones by applying transformations.

- Actions - actions compute some sort of result based on the RDD and then return it to the driver program or save it to an external storage system[8].
One of the most important elements that enable Spark to be fast is lazy evaluation. RDDs can be created at any time and any number of transformations can be applied Spark doesn’t actually compute anything until an action is called. This is a huge advantage when working with big data. Spark can look at an entire chain of transformations from the DAG and compute just the data needed for its final result instead of wasting resources by computing all the data for each step. The way this works is that RDDs also store metadata about the requested transformation operations as well as the ‘parent RDD’ from which the current one is derived (and other dependencies)[8].

Spark offers APIs for Python, Scala, Java and R. Since this project is written fully in Java, all examples provided in this report are in Java as well.

### 2.3.2 Creating RDDs

RDDs can be created either by loading an external dataset or by parallelizing a collection of in-memory objects. The code snippet below provides an example for Spark’s “parallelize” and “textFile” functions for creating RDDs.

**Snippet 2.1: “RDD Creation”**

```java
JavaSparkContext sc = new JavaSparkContext(conf);
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
//RDD created from in memory objects
JavaRDD<Integer> numberRdd = sc.parallelize(numbers);

//RDD created from an external file
JavaRDD<String> textRdd = sc.textFile("file.txt");
```

### 2.3.3 Transformations

Transformations are RDD operations that return a new RDD. As mentioned above, transformation computations are delayed in a lazy fashion until an action is called. The main Spark transformations are filter, map, flatMap, distinct, sample. Spark also has a special class for handling tuples in the form of a PairRDD which supports transformations such as grouping and reducing by key. There are also a variety of transformations that can operate on two RDDs such as union, intersection, subtraction and cartesian product. The small example below illustrates two basic transformations.

**Snippet 2.2: “RDD Transformations”**

```java
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile("file.txt");

lines = lines.filter(line -> line.startsWith("a"));
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
```
As RDDs are derived from one another using transformations, Spark keeps track of all the dependencies between them in something called the **lineage graph**. This is used to compute each RDD on demand and also for lost data recovery [8].

### 2.3.4 Actions

Actions are the final step in the life-cycle of an RDD. They trigger computations and return the results to the driver program or write them to some storage system [8]. The most common actions are take() and collect(), the latter is used to retrieve all the contents of an RDD and the former can be used to retrieve only a subset of all the results in case the entire RDD is too large and does not fit in the memory of a single machine. The example below illustrates the use of a take call which returns the first 100 items in the RDD.

**Snippet 2.3:** "Take Action on an RDD"

```java
JavaSparkContext sc = new JavaSparkContext(conf);
JavaRDD<String> lines = sc.textFile("file.txt");
lines = lines.filter(line -> line.startsWith("a"));
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
List<Integer> result = lineLengths.take(100);
result.forEach(n -> System.out.println(n));
```

### Spark Functions and Closures

Most of the transformations and some of the actions in Spark need the user to specify a function that is used to compute data (for example filter expects a function that returns an element-wise boolean value). The Java API allows the user to pass in lambda expressions is Java 8 or higher is used but the more general approach is to pass a class that implements one of Spark's function interfaces from the org.apache.spark.api.java.function package [8]. The example below illustrates such an implementation:

**Snippet 2.4:** "Java Function Passing"

```java
public class ContainsString implements Function<String, Boolean> {
    private String _parameter;
    public ContainsString(String parameter) {
        this._parameter = parameter;
    }
    public Boolean call(String x) {
        return x.contains(_parameter);
    }
}
```
2. Background and Related Work

```java
JavaRDD<String> lines = sc.textFile("file.txt");
lines = lines.filter(new ContainsString("aaa"));
```

2.3.5 Running on a Cluster

Spark uses a master-slave architecture where the central master is the Spark Driver, which communicates with a large number of executors or workers, each in their own Java process [8]. Spark can run on multiple machines using an external cluster manager like YARN or Apache Mesos (it also has a built-in manager).

![Spark Cluster Architecture](image)

The driver is responsible for transforming the job into tasks and then scheduling those tasks on executor nodes. As mentioned above, all Spark programs follow a basic structure of creating RDDs from some data source, deriving new RDDs via transformations and calling some actions to save the results.
A DAG is thus created from the user program. Under the hood, Spark performs optimizations such as merging several consecutive transformations together and grouping parts of the DAG into stages, each consisting of multiple tasks. The driver then evaluates the available executors and tries to schedule tasks on the best executor considering data placement.

2.4 JSONiq

JSONiq is a declarative language designed to manipulate JSON data. It was developed by Jonathan Robie, Ghislain Fourny, Matthias Brantner, Daniela Florescu, Till Westmann, Markos Zaharioudakis. It is based on XQuery (a W3C standard) from which it borrows several concepts such as the functional language style, the semantics of comparisons and the structure of FLOWR constructs while dropping the peculiarities of XML. Its main design purpose is to allow multiple NoSQL document stores to be queried in a unified and elegant manner.

The main characteristics of JSONiq are [10]:

1. JSONiq is a set-oriented language. It is designed to process sequences of objects (not individual objects one at a time).
2. JSONiq is a functional language. Every language construct is an expression and the result is an evaluation of an expression.
3. JSONiq is a declarative language. Similar to traditional SQL, only the desired results have to be coded while the algorithm and hardware details can be left to an optimizer.
4. JSONiq is designed to handle semi-structured, nested data.

2.4.1 Core Language

JSONiq handles sequences of JSON elements (objects, arrays, strings, numbers, booleans, nulls and other types) which are called items. Thus, any JSONiq expression returns flat sequences of items as below.

Snippet 2.5: "A JSONiq sequence"

```
3.14, "foo", null, -9, { "foo" : "bar" }, [ 3, 4, 5 ]
```

Types

JSONiq handles semi-structured data: while it is a strongly typed language and will warn the user if any type mismatch or illegal operation is performed, it does not require explicit type declarations. If no type is specified, the most general sequence type, `item*`, is used. The item type hierarchy
is illustrated below. The most general subtypes of Item are AtomicItem and JSONItem. Subtypes of AtomicItem include null, string, boolean and numeric types while objects and arrays are subtypes of JSONItem. Further types that are not JSON related can be supported by JSONiq, including types from the XML schema.

**Figure 2.2: JSONiq Item Type hierarchy**

A sequence is an ordered list of items and, as mentioned above, the most general sequence type is \texttt{item*}. A sequence type has two parts, an item type and an arity indicator:

- \* matches sequences of any length (zero or more items)
- + matches non-empty sequences (one or more items)
- ? matches singleton sequences (zero or one item)
- No indicator matches any singleton sequence (one item).

Explicit type declarations for variable initializations are not enforced by JSONiq, if the user does not declare the variable type, the engine defaults to the \texttt{item*} type.

**Basic Expressions**

JSONiq supports all the basic expression types that are to be expected from a functional query language. Users can build sequence types, do arithmetic operations, comparisons, logical operations, call functions and handle objects or arrays. The example below illustrates a few basic expressions combined with a comma operator to generate a single sequence.

**Snippet 2.6: "JSONiq basic expressions"**

\[
1 + 2 * 4 - 5 \text{ idiv } 8 \text{ mod } 2,
\]
Control Flow Expressions and Functions

JSONiq supports several control flow expression types including if and switch along with try/catch blocks and function declarations and calls. The example below illustrates the use of control flow expressions and a built in function call.

**Snippet 2.7: "JSONiq basic expressions"**

```
if (2 + 2 eq 4)
then { "result" : true }
else { "result" : false },

switch ("foo")
  case "bar" return "it's a bar"
  case "foo" return "it's a foo"
  default return "none",

concat("foo", "bar")
```

**[OUTPUT]:** { "result" : true }, "it's a foo", foobar

Other Features and Implementations

JSONiq also supports modules, XQuery-style namespaces, global variables, function libraries and type handling expressions. JSONiq support is now implemented in several frameworks including Zorba, Apache VXQuery and NitrosBase.

2.4.2 FLOWR Expressions and Tuple Streams

FLWOR expressions are “the most powerful JSONiq construct” [10]. They are modeled after XQuery FLWORs and, on a theoretical level, one can say that they are somewhat equivalent to SELECT-FROM-WHERE statements in SQL. There are several clause types that can form a FLWOR expression but all must start with a let or a for clause and the final clause must be a return.
All FLWOR clauses operate with a concept known both in JSONiq and XQuery as a **tuple stream**. Each time, the current clause creates a tuple of "variable bindings (mapping variable names to sequences) which is passed on to the next clause"[10] until the return clause is reached. "The return clause is eventually evaluated for each tuple of variable bindings, resulting in a sequence of items for each tuple"[10].

The clause types are described in more detail below. For most of the examples, we consider a small subsample of "The Great Language Game" dataset [19], version 2014-03-02 with only the 5 entries shown below:

**Snippet 2.8:** "Great Language Game sample"

```json
1 { "guess" : "Latvian", "target" : "Russian", "country" : "AU", "choices" : [ "Lao", "Latvian", "Russian", "Swahili" ], "sample" : "b7df3f9d67cecf259fbca5abcad9774", "date" : "2013-08-20" }
2 { "guess" : "Russian", "target" : "Russian", "country" : "AU", "choices" : [ "Croatian", "Nepali", "Russian", "Slovenian" ], "sample" : "a59d48e99e8a1df7e366c4648095e27", "date" : "2013-08-20" }
3 { "guess" : "Greek", "target" : "Serbian", "country" : "SE", "choices" : [ "Maori", "Czech", "Korean", "Turkish" ], "sample" : "1787b5c79a00b3513ce76847bc1f5b575", "date" : "2013-08-20" }
4 { "guess" : "Serbian", "target" : "Serbian", "country" : "AU", "choices" : [ "German", "Greek", "Kannada", "Serbian" ], "sample" : "0d5b697ebb326b5043ce7fa60a7b968d", "date" : "2013-08-20" }
5 { "guess" : "Serbian", "target" : "Serbian", "country" : "AU", "choices" : [ "Dari", "Serbian", "Sinhalese", "Vietnamese" ], "sample" : "0d5b697ebb326b5043ce7fa60a7b968d", "date" : "2013-08-20" }
```

**FOR Clause**

For clauses enable iteration through sequences. The variable name is bound in turn to each item in the sequence and a new tuple is created[10].

**Snippet 2.9:** "Basic For Clauses"

```
for $x$ in ( 1, 2, 3 )
for $y$ in ( 1, 2, 3 )
return $x + y$

[OUTPUT] : 2, 3, 4, 3, 4, 5, 4, 5, 6
```

Considering a basic query on the confusion dataset sample, the generated tuple streams are illustrated under each instruction. Assume that the dataset is read from a file called "conf-ex.json" in JSON Lines format (one JSON object on each line):

**Snippet 2.10:** "For Clause"

```for $item$ in j$n$ : $parse - json(f$read-text("conf-ex.json"))$
```

```json
1 { $item$ : { "guess" : "Latvian", "target" : "Russian", "country" : "AU", "choices" : [ "Lao", "Latvian", "Russian", ..., ] } }
```
2.4. JSONiq

```json
4 (item : {
  "guess" : "Greek",
  "target" : "Serbian",
  "country" : "SE",
  "choices" : [ "German", "Greek", "Kannada", . . . ] . . . })
5 (item : {
  "guess" : "Serbian",
  "target" : "Serbian",
  "country" : "AU",
  "choices" : [ "Dari", "Serbian", "Sinhalese", "Vietnamese", . . . ] . . . })
return item.guess
```

[OUTPUT]: Latvian, Russian, Czech, Greek, Serbian

The “syntax” for the tuple streams is just for illustrative purposes and the dots at the end are used to shorten the objects. In essence, tuples are just maps from variable names to sequences of items, in case of this query, the variable item gets mapped to each object in the file using the for clause and then a field is accessed and returned in the return clause.

For clauses can also be combined, support multiple variables within the same line and can provide positional variables in the form of an integer index. A theoretical equivalence between JSONiq’s FOR + RETURN queries and SQL’s SELECT FROM queries can be observed[10].

WHERE Clause

Where clauses are used for filtering and are somewhat equivalent to SQL’s ‘WHERE’/‘HAVING’ keywords. “For each incoming tuple, the expression in the where clause is evaluated to a boolean”[10]. If the result is false, the tuple is dropped. Using the same conventions as in the previous subsection, the following query illustrates the resulting tuple streams of the where clause:

Snippet 2.11: "Where Clause"

```json
for item in json.parse(json(f.read.text("conf-ex.json")))
1 (item : {
  "guess" : "Latvian",
  "target" : "Russian",
  "country" : "AU",
  "choices" : [ "Lao", "Latvian", "Russian", . . . ] . . . })
2 (item : {
  "guess" : "Russian",
  "target" : "Russian",
  "country" : "AU",
  "choices" : [ "Croatian", "Nepali", "Russian", . . . ] . . . })
3 (item : {
  "guess" : "Czech",
  "target" : "Czech",
  "country" : "SE",
4 (item : {
  "guess" : "Greek",
  "target" : "Serbian",
  "country" : "SE",
  "choices" : [ "German", "Greek", "Kannada", . . . ] . . . })
5 (item : {
  "guess" : "Serbian",
  "target" : "Serbian",
  "country" : "AU",
  "choices" : [ "Dari", "Serbian", "Sinhalese", "Vietnamese", . . . ] . . . })
```
2. Background and Related Work

The for clause binds the item variable to 5 objects and the where clause drops the tuples that don’t fulfill the filter condition (having the country equal to “AU”). In the end only 3 tuples remain and the return clause returns the “guess” field of each. Where clauses support any type of expression that can return a boolean so multiple conditions can be combined using logic operators:

```java
for $item in json(parse-text("conf-ex.json"))
where $item.choices[[1]] eq "Lao" and $item.country eq "AU"
return $item
```

### LET Clause

Let clauses allow the user to create a new variable (or assign a new sequence to an existing one). The expression in the let clause is evaluated to a sequence and the result is assigned to the variable[10]. The incoming tuple is then extended to include the new variable.

#### Snippet 2.12: "Let Clause"

```java
for $item in json(parse-text("conf-ex.json"))

let $guess := $item.guess, $target := $item.target
```

[OUTPUT]: Latvian, Russian, Serbian

```json
{
  "guess": "Latvian",
  "target": "Russian",
  "country": "AU",
  "choices": ["Lao", "Latvian", "Russian", "Swahili"],
  "sample": "b7df3f9d67cef259fbca95abcad9d774",
  "date": "2013-08-20"
}
```
In the example above, the let clause creates 2 new variables for the guess and target fields of each item variable from the incoming tuple and both variables are added to the tuple.

ORDER BY Clause

Order by clauses are used to sort incoming tuples and are somewhat equivalent to the ORDER BY instructions in SQL.

Snippet 2.13: "Order By Clause"

```plaintext
for $item in json(parse-text("conf-ex.json"))
    order by $item.target descending
return $item.target
```

[OUTPUT]: Serbian, Serbian, Russian, Russian, Czech

Order By clauses can sort by multiple keys ascending or descending and can also handle empty items.
GROUP BY Clause

JSONiq also supports grouping via the Group By clause. The expression in the group clause is evaluated to a grouping key (which must be atomic) and incoming tuples are then grouped accordingly[10]. The Core JSONiq language supports grouping by existing variables as well as declarations of new variables within the group by expression. Consider the following query:

Snippet 2.14: "Group By Clause"

```json
for $i$ in 1 to 5
for $j$ in 1 to 2
    group by $j$, $\text{mod} := \text{i} \mod 2$
    return {
        "$i" : $i$, "$j" : $j$, "$\text{mod}" : $\text{mod}$
    }
```

[OUTPUT]:
```
{
    "$i" : [ 2, 4 ],
    "$j" : 1,
    "$\text{mod}" : 0
},
{
    "$i" : [ 1, 3, 5 ],
    "$j" : 1,
    "$\text{mod}" : 1
},
{
    "$i" : [ 1, 3, 5 ],
    "$j" : 2,
    "$\text{mod}" : 1
},
{
    "$i" : [ 2, 4 ],
    "$j" : 2,
    "$\text{mod}" : 0
}
```

2.5 Related Work

2.5.1 Apache VXQuery

Apache VXQuery [6] is an XML query processor implemented in Java. At the time of this report, it is still in development. Similar to this project, VXQuery is focused on processing large amounts of XML data, specifically in the form of a large number of small XML documents. It uses the Hyracks parallel execution engine (which predates Spark) and Algebrics, a compiler toolbox and implements XQuery Specific parsers and data models on top. It also has JSONiq support.
2.5.2 Document Stores

Document stores are becoming increasingly popular and a large part of the available technologies use JSON or closely related formats when handling data[2][3][9]. Some, like Elasticsearch and MongoDB have their own JSON-based query languages. Elasticsearch is used as a reference to analyze the performance of this engine in section 5 since it seems to best the fastest framework for read operations processing millions of records[20].

2.5.3 Other JSON Processing Languages

JSONiq is of course, closely related to XQuery[11] but more query languages for JSON are increasingly becoming available. Some of these include: JSON Query, N1QL, UNQL. The last two aim to bring a SQL-like syntax for semi-structured data queries. There are also variants of JSONiq adapted to languages such as PythonQL, similar to C#’s LINQ.

2.5.4 Data Models for Parallel Frameworks

While MapReduce and Spark were introduced in this section, several frameworks that build on top of these tools with data models or languages are worth mentioning. Apache Hive is a data warehouse project that aims to bring a unified, SQL-like interface in order to query data stored on HDFS (HiveQL)[1]. The way it works is that it transparently converts SQL queries to MapReduce jobs or even chains of Spark transformations and actions.

GraphX is a Spark API designed to process graphs and “graph-parallel computations”[15]. 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.
Chapter 3

Theoretical Aspects of Parallel JSONiq Queries

This chapter explains the theory used to map tuple streams to RDDs and JSONiq FLWOR clauses to Spark transformations and actions in order to enable the execution of JSONiq queries on top of Spark.

3.1 Tuple Streams to RDDs

As presented in section 2.4.2, the central concept behind FLWOR expressions is the tuple stream. Each tuple is essentially a map between variable names and sequences of items and the tuple stream gets passed down from one FLWOR clause to the next, each of them processing the incoming tuple stream according to their own semantics. Since the Spark Java API supports user-defined types in the generic RDD classes, RDDs that handle a tuple object type can be created.

Thus, a JSONiq FLWOR query is a set of clauses that receive and output streams of tuples. This can be mapped to a set of RDD transformations (potentially containing RDD creations as well) executed on RDDs that contain maps of variable names and sequences encapsulated in tuple objects. The next section presents an equivalent set of Spark transformations for each of the FLWOR clause types.

3.2 FLWOR Clauses to Spark Transformation

In order to run JSONiq FLWOR queries on top of Spark, a mapping between each clause type and a certain set of transformations and actions must be developed:
3. Theoretical Aspects of Parallel JSONiq Queries

3.2.1 For Mapping

The for clause in JSONiq is used for iteration through a sequence. Each incoming tuple is extended to a new set of tuples, each containing all the variables from the original tuple alongside one of the values of the new variable in the current for clause, created by iterating through the current sequence of items in the for clause. Conceptually, this can be achieved using Spark’s `flatMap()` function which produces a flat sequence of elements by taking each input element and mapping it to one or more elements and flattening the end results. The for clause semantics can be emulated on Spark by taking each incoming tuple and mapping it to a sequence of new tuples, each extending the original tuple with one new key-value pair derived from the iteration over the current for clause generated values. Figures 3.1 and 3.2 illustrates an example in order to better understand the equivalence:

**Figure 3.1:** Tuple Streams for 2 for clauses

```plaintext
for $i$ in 1 to 3
    for $j$ in 1 to 3
        return $i + j$
```

**Figure 3.2:** Spark’s `flatMap()`

```plaintext
Input Element 1
Input Element 2
```
3.2. FLWOR Clauses to Spark Transformation

The special case is when the for clause is the very first of the FLWOR expression. In that case, there are is no input tuple stream and thus no flat-map is required.

3.2.2 Let Mapping

The let clause in JSONiq is used to declare a new variable and assign a sequence to it. Thus, the let clause simply extends each incoming tuple to include the new variable alongside the previously existing ones. This can be achieved in Spark by using the map() function to extend each tuple with the new pair of variable name and value (see Figures 3.3 and 3.4).

Figure 3.3: Tuple Streams for a let clause

```plaintext
for $i$ in 1 to 3
  let $j = 3 * 4$
return $i + j$
```

OUTPUT: 13 14 15

Figure 3.4: Spark’s map()

Unlike the for clause where the variable can only have single items as the current value, the let clause can assign an entire sequence of items at once.

Variable Redeclaration

Both the let and the for clauses support variable redeclaration. The variable will simply have the last value or sequence of values that was assigned to it.
3. Theoretical Aspects of Parallel JSONIQ Queries

Prior assignments will continue to exist as hidden variables which are not accessible to the user.

3.2.3 Where Mapping

The where clause is used to filter items in JSONiq, removing the tuples which don’t match the filter criteria from the stream. This can be done in Spark by using the filter() function and passing an equivalent filter condition (see figures 3.5, 3.6).

![Figure 3.5: Tuple Streams for a where clause](image)

![Figure 3.6: Spark’s filter()](image)

3.2.4 Order By Mapping

The order-by clause is used to sort items in JSONiq by rearranging the incoming tuples based on some sort key. This cannot be achieved in Spark using a single transformation, several operations are necessary:

1. Firstly, a `mapToPair()` call is necessary, mapping each tuple to a pair with the first component being the sort key (which can be composite) and the second, the tuple itself.
2. Having mapped the tuple RDDs to PairRDDs, Spark can sort them using the `sortByKey()` function.

3. After the sorting is performed, the mapping operation that was performed in step 1 needs to be reversed in order to preserve the format of the RDDs for the next clause. This can be achieved by calling a `map()` on the PairRDDs and keeping only the original object without the key.

Figure 3.7 illustrates an example. Consider an initial sequence of 4 JSON objects that is read from a file by the first for clause.

3.2.5 Group By Mapping

Group-By in JSONiq is similar to the group clause in SQL and can be used to group objects together based on a specified key. Similar to order-by, in order to achieve this in Spark, multiple operations have to be performed:

1. Like in order by, a `mapToPair()` call is necessary, mapping each tuple to a pair with the first component being the group key (which can be composite) and the second, the tuple itself.

2. Having mapped the tuple RDDs to PairRDDs, Spark can group them using the `groupByKey()` function.
3. Theoretical Aspects of Parallel JSONiq Queries

3. After the grouping is performed, the mapping operation that was performed in step 1 needs to be reversed in order to preserve the format of the RDDs for the next clause. This can be achieved by calling a map() on the PairRDDs and keeping only the original object without the key. The final step calls map() and linearize the results. Each group key will have a corresponding list of tuples. This list of tuples must be linearized to a single tuple by assigning to each variable the merged sequence of values from all the tuples in the sequence.

Figure 3.8: Tuple Streams for a group-by clause

Figure 3.9: Equivalent Spark transformations for group-by

3.2.6 Return Mapping

The return clause ends a JSONiq FLWOR query by returning the desired results. This can be achieved in Spark by calling map() using a function which is equivalent to the return clause expression. If the results must be returned immediately, an action call is necessary, namely collect().
### 3.2.7 Summary and Example

A summary of the mappings from clauses to transformations is presented below:

<table>
<thead>
<tr>
<th>FLWOR Clause</th>
<th>Spark Transformation / Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>flatMap() - map each incoming tuple to a set of tuples, each extending the original one with one new key-value pair</td>
</tr>
<tr>
<td>let</td>
<td>map() - extend each incoming tuple with the new variable name to sequence of items pair</td>
</tr>
<tr>
<td>where</td>
<td>filter(condition)</td>
</tr>
</tbody>
</table>
| order by key | 1. mapToPair() - map each tuple to a pair with the sort key and the tuple itself  
2. sortByKey()  
3. map() - map back from pairs to tuples only |
| group by key | 1. mapToPair() - map each tuple to a pair of group key and the tuple itself  
2. groupByKey()  
3. map() - map back from pairs to tuples only and linearize the results |
| return       | map() + collect()/take() |

Having presented a mapping between the FLWOR clause types and sets of Spark transformations, it is useful to review an example of a JSONiq query and an equivalent Java code snippets to see the transformations and the final results. Consider the input file with the 5 objects presented in the previous chapter, section 2.4.2. For this example, a JSON parser is used to store tuples in JSON format. This is simply a code snippet written manually to perform Spark operations that produce the same result as the JSONiq query. The goal of this project is to automate this.

**Snippet 3.1:**

```java
for $item in json:parse-json(f:read-text("conf-ex.json"))
order by $item.target
return $item.target
```

[OUTPUT]: Czech Russian Russian Serbian Serbian
3. Theoretical Aspects of Parallel JSONIQ Queries

```java
// read the json file
JavaRDD<String> stringRDD = sc.textFile(url);

// create tuple rdd
JavaRDD<Item> objectRDD = stringRDD.map(s ->
        parser.getItemFromObject(new JSONObject("{ "i": " + s + "}")));

// run order by
// step 1
JavaPairRDD<String, Item> pairRDD =
    objectRDD.mapToPair(o -> new
        Tuple2(o.getItemByKey("i").getItemByKey("target")
            .getStringValue(), o));

// step 2
pairRDD = pairRDD.sortByKey();
// step 3
objectRDD = pairRDD.map(t -> t._2);
// output
JavaRDD<String> result = objectRDD.map(o ->
    o.getItemByKey("i").getItemByKey("target")
        .getStringValue().toString());
System.out.println(result.collect());
```

[JAVA OUTPUT]: Czech Russian Russian Serbian Serbian
Chapter 4

Project Implementation

4.1 General Architecture

The engine follows a somewhat traditional compiler architecture. It is heavily inspired by the Zorba XQuery Processor. There are several layers that enable the engine to take JSONiq query as input, translate it, execute it and produce results. Figure 4.1 presents a diagram of the architecture.

1. The first step, as in any compiler, is to use a lexer and parser in order to transform the query text into an Abstract Syntax Tree (AST). In this project, a grammar file for JSONiq was created and the lexer and parser were automatically generated from the grammar, using the ANTLR v4 framework.

2. The next step is to translate the AST into a tree of expressions, using classes defined for each expression type.

3. The next step is the code generation. In this case, the expression tree is converted to a tree of Java runtime iterators which encompass the logic of each operation and can be executed in order to produce sequences of items.

4. The final step is to execute the runtime iterators (either locally on a single node or by manipulating RDDs on top of Spark) and collect the results.
4.2 Compiler

4.2.1 Lexer and Parser

The first step in the process of executing the JSONiq query is to parse the input query. For this task, the ANTLR framework[17] is used. ANTLR is a parser generator than can generate a parser capable of building and walking parse trees from a grammar file. The ANTLR v4 framework uses ALL(*) [16] parsing and is also capable of generating an AST in addition to the lexer and parser.

Thus, the first part of the project was to take the official JSONiq Core .ebnf grammar file and convert it to the .g4 format which is supported by ANTLR. Using the grammar file in .g4 format and the ANTLR binaries, the basic front-end of the compiler can be generated with one command. ANTLR generates a JSONiq lexer, parser and a base class implementing the visitor pattern which can be used to generate expression trees from the AST.

4.2.2 Expression Tree Generation

As in most compilers, the AST cannot really be used directly. The next step is to take the AST and convert it into an expression tree. Using the base visitor class generated by ANTLR in the previous step, an expression tree can be generated.

The general rule in JSONiq is that every syntactic building block is an expression. The only slightly counter-intuitive construct is the FLWOR. The whole FLWOR is an expression, containing multiple clauses. It is thus natural to have a base abstract class that can encapsulate basic functionality for any expression or clause. The snippet below contains a partial implementation of the base abstract class. It contains support for a visitor pattern implementation, basic serialization and functionality for retrieving children.
4.2. Compiler

Snippet 4.1: "Base class for all expressions and clauses"

```java
public abstract class ExpressionOrClause {

    // Visitor pattern implementation
    public abstract <T> T accept (AbstractExpressionOrClauseVisitor<T> visitor, T argument);

    // used to retrieve children
    public abstract List<ExpressionOrClause> getDescendants(boolean depthSearch);
    public List<ExpressionOrClause> getDescendants() {
        return getDescendants(false);
    }

    // serialization
    public abstract String serializationString (boolean prefix);

    protected ExpressionOrClause () {}

    // ...
}
```

4.2.3 Static Context

Each expression’s static context contains information required in the static analysis phase [11]. This includes, in-scope variables and function declarations, statically known namespaces and collations, ordering modes. For this project, the only concern is to identify the in-scope variables and their declared types since user-defined functions, modules and namespaces are out of scope. The Appendix contains a complete list of supported features.

Thus, a static context contains a map between variable names and sequence types. Each expression has its own static context. Consider the example below:

Snippet 4.2: "Static Context Example"

```java
for $i in 1 to 5
    for $j in 1 to $i
        return $j
```

[OUTPUT]: 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5

Looking at what the expression tree would look like, the static context of each expression can be put into perspective:
In the figure above, the expression tree of the query presented above is illustrated along with each expression’s static context (individual FLWOR clauses and their variable initializations are not expressions and thus, have no static context) in the upper-right corner. Assuming that there are no declared variables before this query and that we start with a root static context, the way the contexts are generated becomes clear. The first for clause and its children expressions have no variables in the static context. This first clause initializes the variable $i$, since no type is explicitly declared, the engine defaults to item* which is the most general sequence type, as explained in Chapter 2. This new variable is passed down to the static contexts in the following clauses. The second for clause and all of its children now contain the variable $i$ in their static context. Since $i$ is referenced in the range expression of the second for clause, it must be present in the current static context as a declared variable, otherwise an error would be thrown. The second clause also initializes the $j$ variable which gets passed down to the next clause which will contain both variables that were declared above.

Since every single expression in the query has its own static context, keeping multiple copies of all the variables can be costly to memory usage. A better approach is to chain the static contexts together so that each one contains a reference to the previous one and does not duplicate any variables. This is common practice in XQuery engines and thus, this strategy was adopted in this project as well. An sample of the static context implementation is presented below:

```
Snippet 4.3: "Static Context Implementation (partial)"

public class StaticContext {
    public static context getParent() {
        return _parent;
    }
}
```
4.2. Compiler

```java
protected Map<String, SequenceType> getInScopeVariables() {
    return _inScopeVariables;
}

public StaticContext(StaticContext parent) {
    this._parent = parent;
    this._inScopeVariables = new HashMap<>();
}

public SequenceType getVariableSequenceType(String varName) {
    if (_inScopeVariables.containsKey(varName))
        return _inScopeVariables.get(varName);
    else if (_parent != null)
        return _parent.getVariableSequenceType(varName);
    else
        throw new SemanticException("Variable " + varName + " not in scope");
}

public void addVariable(String varName, SequenceType type) {
    this._inScopeVariables.put(varName, type);
}

// ...
```

The final piece is the generation of the static context. Given a generated expression tree, the static contexts for all expressions need to be generated. As presented in section 4.2.2, the base class for all expressions has an abstract accept method which is meant to facilitate a visitor pattern implementation. Given the nature of expression trees it is often natural for compiler implementations to make extensive use of the visitor pattern. This project includes an abstract visitor with a multitude of methods, each responsible for visiting an unique expression subtype and the default visiting action being to simply continue to visit all descendants.

The visitor starts at the top level expression and then moves through all of the children passing along the current static context while doing three things:

1. For any expression that it visits, it sets the static context to be equal to the currently generated one.

2. For any variable reference, it checks that the variable name is present in the current static context, otherwise it throws an error (at compile
4. Project Implementation

3. For any variable declaration it creates a new static context containing the new variable and sets the previously existing static context as parent.

4.2.4 Item Hierarchy

As explained in section 2.4.1, variables and constants have sequence types. Each sequence type is comprised of a item type and an arity. The item type hierarchy is implemented using a base abstract class and a series of subtypes that implement the abstract methods and override the boolean methods accordingly. A partial implementation of the base class is presented below:

**Snippet 4.4:** "Static Context Visitor Implementation (partial)"

```java
public abstract class Item implements Serializable {

    public boolean isArray() { return false; }
    public boolean isObject() { return false; }
    public boolean isAtomic() { return false; }
    public boolean isNumber() { return false; }
    public boolean isString() { return false; }
    public boolean isBoolean() { return false; }
    public boolean isNull() { return false; }
    public boolean isInteger() { return false; }
    public boolean isDouble() { return false; }
    public boolean isDecimal() { return false; }

    public abstract Item getItemAt(int i) throws UnsupportedOperationException;
    public abstract Item getItemByKey(String s) throws UnsupportedOperationException;
    public abstract void putItemByKey(String s, Item value) throws UnsupportedOperationException;
    public abstract List<String> getKeys() throws UnsupportedOperationException;
    public abstract int getSize() throws UnsupportedOperationException;
    public abstract String getStringValue() throws UnsupportedOperationException;
    public abstract boolean getBooleanValue() throws UnsupportedOperationException;
    public abstract double getDoubleValue() throws UnsupportedOperationException;
    public abstract int getIntegerValue() throws UnsupportedOperationException;
    public abstract BigDecimal getDecimalValue() throws UnsupportedOperationException;
}
```
As explained in Chapter 2, section 4, JSONiq supports a multitude of types. This project implements a subset comprised of the following types:

- **Atomics**: integer, double, decimal, boolean, string, null
- **JSON items**: JSON array, JSON object

### 4.3 Runtime Iterators

The final step in translating JSONiq queries is to convert the expression tree into a tree of runtime iterators. Unlike traditional compilers for general purpose, imperative languages which convert the intermediate representation into assembly instructions, the runtime iterators are responsible for handling the execution of operations which get converted to Java bytecode.

Each runtime iterator can walk through a sequence of items and return it, one item at a time. Given a tree of iterators, a call to retrieve an item will cause the current iterator to recursively pull an item from the child iterators and then process the resulting items according to its own semantics in order to produce the result. An example of a basic iterator tree for the arithmetic operation $1+2+3$ is presented below:

![Figure 4.3: Example of a expression tree and static context](image)

Considering the additive expression above, calling on the iterator to return the next item makes calls to the children that propagate recursively. The integer iterators simply return the value and the additive iterators compute the sum of the 2 child values and return the result.

The interface of the runtime iterator is as follows:
4. Project Implementation

**Snippet 4.5: "Runtime Iterator Interface"**

```java
public interface RuntimeIteratorInterface extends Serializable {
    void open(DynamicContext context);
    void close();
    void reset(DynamicContext context);
    boolean hasNext();
    Item next();
}
```

- The `open` method opens the current iterator and passes a dynamic context.
- The `close` method closes the current iterator.
- The `hasNext` method checks if the current iterator has any items to iterate through in the current sequence.
- The `next` method returns the next item in the sequence.
- The `reset` method is used to reset the sequence iteration and passes a new dynamic context.

For this project, the runtime iterators can be split into 2 categories: local runtime iterators, which perform operations locally and Spark-enabled runtime iterators which require Spark calls and execute the logic on a cluster (see section 4.4). Local runtime iterators have been implemented for the following features of JSONiq (the appendix contains a complete list of supported and unsupported features):

- Primary Expressions: integer, double, decimal, string, boolean and null items as well as JSON arrays and object items.
- Operational Expressions: arithmetical expressions (+, -, div, idiv, mod, *), logical operations (and, or, not, comparisons).
- General expressions: control flow (if, switch), comma expressions and certain library function implementations.

### 4.3.1 Translator and Dynamic Context

As explained in section 4.2.3, the static context contains a mapping between variable names and sequence types (alongside other information) and is used in the static analysis phase. At runtime, a dynamic context is needed to keep the variable values. Similar to the static context which maps variable names to sequence types, the dynamic context keeps a mapping between variable names and actual sequences of items.
The translation of expression trees into runtime iterators is done, just as in the case of the static context (see section 4.2.3), using the visitor pattern. A visitor is used to traverse the expression tree and return a runtime iterator for each expression. One change that needs to be made is to convert n-ary expressions into a tree of binary runtime iterators.

4.4 Spark-Enabled Iterators

Unlike local runtime iterators, which compute a sequence of items locally, based on the iterator semantics, FLWOR runtime iterators have to run on top of Spark. As explained in section 2.3, Spark makes use of partitioned RDDs and chains of lazily computed transformations in order to produce results. In order to facilitate the underlying use of RDDs, the base abstract class of all runtime iterators needs to be extended with two additional methods:

1. `boolean isRDD()` - used to differentiate between local runtime iterators and Spark-enabled iterators which use RDDs underneath.
2. `JavaRDD<Item> getRDD()` - used to retrieve the underlying RDD of the current runtime iterator. `JavaRDD` is the generic wrapper class exposed by the Spark Java API and `Item` is the base class of all item types presented in section 4.2.4. For any local runtime iterator, this method throws an appropriate exception. For Spark-enabled iterators, the method will take the parent RDD (if any), apply a chain of transformations according to the current iterator semantics and return the resulting RDD. As explained in section 2.3, no computations are done at this point, just a reference to the RDD is returned until an action is called.

The external interface and basic flow of all iterators is the same, no matter if they are Spark-enabled or not. The user opens the iterator, then uses `hasNext()` and `next()` to iterate through the generated sequence of items and then closes the iterator. The figure below presents the underlying functionality for a Spark-enabled iterator. When `next()` is called for the first time, the method calls `getRdd()` in order to apply the required Spark transformations according to the semantics of the iterator and retrieve an updated pointer to the current RDD. Afterwards, the RDD is collected via an action and the resulting sequence of items is returned. Finally, the iterator can walk through this newly generated sequence normally. The lazy functionality of Spark is thus maintained. Iterators can chain and transform RDDs with no computations being performed until `next()` is called for the first time.

Thus, we have a complete hierarchy of runtime iterators. The base functionality of opening and closing iterators can be shared among all iterators. The following subtypes are the local runtime iterators and the Spark-enabled ones. Given the architecture presented above, the `next()` and `isRDD()` is
identical for all Spark-enabled iterators and only the `getRDD()` method has to be implemented distinctly for each type of iterator in order to reflect its semantics and apply the correct chain of transformations. Figure 4.4. presents the hierarchy in more detail.

### 4.4.1 Spark-Enabled Function Iterators

Before this engine can process JSONiq FLWOR queries on top of Spark, it must be able to read and write data from a cluster running Spark (most likely on top of HDFS). In order to achieve this, 2 function iterators were implemented:

1. The `json-file()` function is a JSONiq wrapper for the Spark text-file function presented in section 2.3. It takes a string argument representing a file path and an optional integer argument indicating the number of partitions. As explained in the previous paragraph, only the `getRDD()` method needs to be implemented, all other functionality being shared among Spark-enabled iterators. In this case, the iterator calls Spark’s `text-file()` function internally using the current path argument. This function can automatically handle both URIs pointing towards local files and HDFS files. This call returns a `JavaRDD<String>` reference. The second step is to map the strings to JSON objects using Spark’s `mapPartitions()` function and passing a mapper as argument. For this project, the `org.json` parser[4] is used to convert the strings to JSON objects and thus return a `JavaRDD<Item>` reference. Snippet 4.7 illustrates an example.
2. Similarly to the previous example, a JSONiq wrapper for Spark’s `parallelize()` function is desirable. A function with an identical name was implemented for this JSONiq engine. It takes any number of items as parameters and uses the `parallelize()` call to create a distributed RDD. The snippet below illustrates an example:

```
Snippet 4.7: "A parallelize call"
```n
```
parallelize((
  { "a" : "b" },
  { "c" : "d" },
  [ 1 , 2 ],
  3)
```

```
[OUTPUT]: ( 
  { "a" : "b" },
  { "c" : "d" },
  [ 1 , 2 ],
  3)
```

More functions can be added in the future. For example, a `text-file` function that simply reads text lines and creates string RDDs without any JSON parsing could be useful.
Figure 4.5: Runtime Iterator hierarchy. Base class provides common functionality for open and closing iterators. All other methods are abstract. All LocalRuntimeIterator implementations return false when isRDD() is called and thrown an exception when getRDD() is called. All operational, primary, and control flow iterators extend LocalRuntimeIterator. SparkRuntimeIterator is the base type for all Spark-enabled iterators, the next() and isRDD() functionality is identical for all subtypes. FLWOR iterators and certain function iterators extend this class.
4.5 FLWOR Iterators

All of the previously presented features enable the final step of this project, the implementation of the FLWOR expressions on top of Spark. Recalling the theory in section 2.4, a FLOWR is an expression composed of a list of clauses that must end with a return. While the entire FLWOR itself is an expression which returns a sequence of items (and works with a JavaRDD<Item> reference internally), individual clauses work with tuple streams which are basically mappings between variable names and sequences items.

4.5.1 Local FLWOR Implementation

For this project, the goal is to enable JSONiq FLWOR queries execution on top of Spark. There are of course, certain scenarios where local, Spark-independent implementations of the FLWOR clauses might be useful but they are out of the scope for this project and are NOT supported. Considering the example in snippet 4.9.

Snippet 4.8: "Local and Spark-enabled clauses"

```java
for $i in json-file("/queries/conf-ex.json")
for $j in 1 to 2
return $i.target
```

[OUTPUT]: (Russian, Russian, Czech, Serbian, Serbian,
            Russian, Russian, Czech, Serbian, Serbian)

The first for clause maps the variable to the JSON objects generated by the json-file() function presented in section 4.4.1. This is a Spark-enabled clause that will use RDDs and transformations underneath, the file can very large. The second call maps the variable j to the values generated by a local range expression which in general would generate a much smaller amount of values to iterate through. In this case, the engine simply wraps the results into an RDD and forces the use of Spark for this clause as well (see snippet 4.10):

Snippet 4.9: "Creation of RDDs from RHS expressions"

```java
// used to generate initial RDD for start LET/FOR
protected JavaRDD<Item>
    getNewRDDFromExpression(RuntimeIterator expression){
        JavaRDD<Item> rdd;
        // if the RHS expression of the clause generates an RDD
        just return the reference – this is lazy
        if(expression.isRDD())
            rdd = expression.getRDD();
        // else, compute the results locally, then use the spark
context to parallelize the results
```
4. Project Implementation

```java
else {
    List<Item> contents = new ArrayList<>();
    expression.open(this._currentDynamicContext);
    while (expression.hasNext())
        contents.add(expression.next());
    expression.close();
    rdd = SparkContextManager.getInstance()
        .getContext().parallelize(contents);
}
return rdd;
}

4.5.2 FLWOR Tuples and Closures

While all iterators must return sequences of items and this is applies to FLWOR iterators as well, the individual clauses make use of tuples and tuple streams. All of the FLWOR clause iterator implementations make use of closures. Closures are function arguments used by most of Spark’s transformation methods. They are serialized and sent to the cluster and must implement one of the Spark Java API “Function” interfaces [8]. The code snippet below presents an example of a closure used for the Where clause. It takes tuples and return booleans generated by the logical condition inside the where clause:

Snippet 4.10: "Closure used for Where Clause"

```java
// indicates that the function returns booleans from Flwor tuples
public class WhereClauseClosure implements Function<FlworTuple, Boolean> {
    private final RuntimeIterator _expression;

    public WhereClauseClosure(RuntimeIterator expression) {
        this._expression = expression;
    }

    @Override
    // for each tuple, return a boolean
    public Boolean call(FlworTuple v1) throws Exception {
        _expression.open(new DynamicContext(v1));
        Item result = _expression.next();
        _expression.close();
        return result.getBooleanValue();
    }
}
```

In this instance, the runtime iterator corresponding to the condition of the where clause is also serialized and sent to the cluster alongside the entire closure.
4.5.3 For and Return Clauses Implementation

As explained in section 3.2.1, the for clause is used to iterate through a sequence of item, and can be translated conceptually to a flat-map operation from Spark. There are two possible use cases for the for clause:

1. **The for clause is the first clause of a FLWOR expression.** If the current clause is the very first one in the FLWOR expression, then it has no prior tuple stream to process and must create a new one, by mapping the variable name in turn, to each value in the sequence of items that it iterates through, each time creating a new tuple. The clause expects an iterator that generates an RDD from the right hand side assignment expression, if that is not the case, it simply wraps the results into an RDD. The for iterator takes that Item RDD reference generated by the assignment expression, and then it calls Spark’s `map()` function and passes down a closure implementation that simply creates a new FLWOR tuple and adds the variable name with the current Item value into it. In this case it is not necessary to use `flatMap()`, it can be simplified.

2. **The for clause is not first clause of a FLWOR expression.** If the current clause is not the very first one in the FLWOR expression, the clause must process the incoming tuple stream and for each incoming tuple, use `flatMap()` to generate a list of new tuples by iterating through the values produced by the assignment expression and for each of them creating a new tuple (identical to the incoming tuple at first) and adding the current [variable name:value] pair to it.

Code snippet 4.12 presents a partial implementation of the ForClauseSparkIterator class as well as a closure example.

```
// part of the ForClauseSparkIterator class
@override
public JavaRDD<FlworTuple> getTupleRDD() {
    //...
    if (this._rdd == null) {
        //....
        JavaRDD<Item> initialRdd = null;
        // if it's a start clause
        if (this._previousClause == null) {
            initialRdd =
                this.getNewRDDFromExpression(assignmentExpression);
            this._rdd = initialRdd.map(new InitialForClauseClosure(variableReference));
        } else {
            // if it's not a start clause
```
4. Project Implementation

```java
this._rdd = this._previousClause.getTupleRDD();
this._rdd = this._rdd.flatMap(new ForClauseClosure(assignementExpression, variableReference));
}
return _rdd;
}

// InitialForClauseClosure implementation
public class InitialForClauseClosure implements Function<Item, FlworTuple> {
    private final String _variableName;

    public InitialForClauseClosure(String variableName) {
        this._variableName = variableName;
    }

    @Override
    public FlworTuple call(Item v1) throws Exception {
        FlworTuple result = new FlworTuple();
        result.putValue(_variableName, v1, true);
        return result;
    }
}
```

Return Clauses

Return clauses map the incoming RDD of tuples back into an RDD of items by applying the return expression iterator via a `map()` function. As explained above, this returns the final RDD reference for the entire FLWOR expression, no computations are being done when `getRDD()` is called. In fact, the iterator of the entire FLWOR expression implements this method by simply calling the `getItemRDD()` method of the return clause which will recursively call on all clauses to return their own RDD reference and apply their own chains of transformations in the process.

```
Snippet 4.12: "FLWOR expression iterator"
public class FlworExpressionSparkRuntimeIterator extends SparkRuntimeIterator {
    // ...
    @Override
    public JavaRDD<Item> getRDD() {
        // simply return the RDD reference of the return clause
        return _returnClause.getItemRDD();
    }
```
Positional Variables in For-Clauses

JSONiq for-clauses also support positional variables. These create an extra variable that hold the index of the current item within the sequence. This is NOT supported in this project. The reason is that Spark does not support any way of implementing a globally synchronized counter. While accumulators provide the ability to write from all the nodes of the cluster into a single location, they are write-only. Implementing global synchronization for read-write variables across Spark-enabled cluster nodes is outside the scope of this project.

4.5.4 Where Clause Implementation

Where clauses are used to filter tuple streams and have a direct Spark correspondent in the `filter()` function. The iterator retrieves the RDD reference from the previous clause and calls `filter()` while passing along a closure containing the corresponding runtime iterator of the filter condition.

Snippet 4.13: "Where-clause Iterator Implementation"

```java
@override
public JavaRDD<FlworTuple> getTupleRDD()
{
    if (this.rdd == null) {
        RuntimeIterator expression =
        this._children.get(0);
        if (this._previousClause != null) {
            this.rdd = _previousClause.getTupleRDD();
            this.rdd = this.rdd.filter(new WhereClauseClosure(expression));
        } else {
            // ...
        }
    }
    return _rdd;
}
```

4.5.5 Let Clause Implementation

Let clauses are used to assign a sequence of items to a new variable. Conceptually, this can be done using the `map()` function in Spark. There are two scenarios for the let-clause:

1. The let-clause is not the first clause in a FLWOR expression. In this case the iterator calls `map()` on the previous RDD and passes a closure that simply extends each incoming tuple by adding a new [vari-
4. **Project Implementation**

able name:value pair to it. The variable name and right-hand-side assignment expression also get passed to the closure, serialized and evaluated on the cluster.

2. **The let-clause is the first clause in a FLWOR expression.** In this case, similarly to the initial for-clause presented in section 4.5.3, the initial RDD must be created, in this case by creating a single tuple containing a pair mapping the current variable name to the entire sequence of items generated by the assignment expression.

Snippet 4.14: "Let-clause Iterator Implementation"

```java
@Override
public JavaRDD<FlworTuple> getTupleRDD() {
    if (this._rdd == null) {
        // ...
        // if it’s not a start clause
        if (this._previousClause != null) {
            this._rdd = this._previousClause.getTupleRDD();
            this._rdd = this._rdd.map(new LetClauseMapClosure(variableName, expression));
        } else {
            // if it’s a start clause
            rdd = this.getNewRDDFromExpression(expression);
        }
    }
    return _rdd;
}
```

**Assignments of Large Sequences to Single Variables**

While the engine theoretically allows the use of any expression in the right-hand-side assignment of a let-clause, assigning a very large sequence of items to a single variable can be problematic. This is because Spark is not designed to handle large single items, but primarily it’s for very large numbers of small items. This would have to be implemented very differently, but for the moment, it’s a limitation of this engine. For this reason, FLWOR clauses starting with a let clause that binds a variable to an RDD (presumably a large source of data) are not supported.

**4.5.6 Order By Clause Implementation**

The order-by-clause is used to sort tuples. In order to achieve the same results in Spark, the 3 steps presented in section 3.2.4 must be implemented.
Flwor Key

Since order-by clauses support sorting by multiple fields, a class encompassing a composite key must be created. In our case, any composite key is essentially an ordered list of JSON items. The FlworKey encapsulates this functionality as well as support for comparing two keys by running appropriate comparisons between individual items in each key. This class can be used in both order-by and group-by clauses. For both clause types, only atomic items are allowed inside the composite keys.

4.5.7 Order-by Clause

Using the above mentioned implementation of the composite key the order-by-clause is implemented as follows:

1. The incoming tuple stream is mapped to a pair RDD with the first element being a composite key and the second element being the current tuple itself. The composite key is generated inside the closure passed down to the `mapToPair()` call by executing all of the expression iterators inside the order-by-clause and collecting the resulting items into the composite key, in order.

2. Given the new pair RDD, a `sortByKey()` call is necessary in order to sort the pairs by the key (in this case, exactly the composite key). The closure passed to this call simply handles the comparison of two keys while also handling other sorting related commands such as selecting the order to be ascending/descending or where the empty items go.

3. Having sorted the tuples, the final step is to map the pair RDD back to a normal RDD comprised of tuples since the composite keys are not needed anymore.

The snippet below presents a partial implementation of the order-by-clause.

snippet 4.15: "Order-by-clause Iterator Implementation"

```java
@Override
public JavaRDD<FlworTuple> getTupleRDD() {
    if (this._rdd == null) {
        this._rdd = this._previousClause.getTupleRDD();
        // map to pairs - ArrayItem [sort keys], tuples
        JavaPairRDD<FlworKey, FlworTuple> keyTuplePair =
            this._rdd.mapToPair(new OrderByMapToPairClosure(this._expressions,
            _isStable));
        // sort by key
        keyTuplePair = keyTuplePair.sortByKey(new OrderByClauseSortClosure(this._expressions,
            _isStable));
        // map back to tuple RDD
    }
    return this._rdd;
}
```
4. Project Implementation

```java
this._rdd = keyTuplePair.map(tuple2 -> tuple2._2());
return _rdd;
}
```

4.5.8 Group By Clause Implementation

Similarly to order-by, the group-by-clause also makes use of composite keys and follows a similar strategy (see section 3.2.5):

1. The first step is to map the incoming tuple RDD to a pair RDD of keys and tuples, just like in order-by. Each of the group-by expression iterators is executed and the results collected in the composite key. The difference here is that group-by also allows for new variables to be declared inside the group-expressions. These variables need to be added to the tuples within the same closure.

2. Having the pair RDD, a call to Spark’s `groupByKey()` will group all pairs by the key, using the comparison functionality inside the FlworKey class. This will result in pairs of keys and sequences of tuples.

3. The final step is to return the RDD to the expected format. In order to achieve this, for each unique group key in the pair RDD, the list of tuples must be linearized into one single tuple as follows:

   - For any variable that was part of the group key, that single value (sequence of items) is retained.
   - For any variable that was not part of the group key, all the values from the current list of tuples are merged together into a single sequence of items.

**Snippet 4.16:** "Group-by-clause Iterator Implementation"

```java
@Override
public JavaRDD<FlworTuple> getTupleRDD() {
  if (_rdd == null) {
    _rdd = this._previousClause.getTupleRDD();
  // map to pairs — ArrayItem [sort keys], tuples
    JavaPairRDD<FlworKey, FlworTuple> keyTuplePair =
      this._rdd
        .mapToPair(new GroupByToPairMapClosure(_variables));
  // group by key
    JavaPairRDD<FlworKey, Iterable<FlworTuple>> groupedPair =
      keyTuplePair.groupByKey();
  // linearize iterable tuples into arrays
    this._rdd = groupedPair.map(new GroupByLinearizeTupleClosure(_variables));
```
4.6 Additional Components

4.6.1 Testing Framework

In order to verify the correctness of the implementation, this project includes several types of tests. However, like in most compilers, writing manual unit tests for things such as the AST and expression tree validation is very slow. In order to address this, the testing strategy shifted more towards end to end tests. An annotation framework was developed, in order to allow for the quick testing using actual JSONiq queries. Each query file has to begin with a JSONiq comment containing one of the following annotations:

- **ShouldParse/ShouldNotParse** - used for testing of the grammar file and the ANTLR generated parser and AST. The test fails if the program parses when not expected to or when it doesn’t parse but the annotation says it should.

- **ShouldCompile/ShouldNotCompile** - used for testing the static analysis phase (expression tree generation, static context creation).

- **ShouldRun/ShouldCrash** - used for end to end testing, the program is parsed, runtime iterators are created and executed. The results can also be checked by using a secondary annotation, **Output**, followed by the expected output. The snippet below presents an example.

  **Snippet 4.17:** "Annotated test - should run and output the expected results"

  ```json
  (: J I Q S: ShouldRun; Output="( 5 , 7 , 9 , 11 , 13)" :)
  for $i in parallelize((1,2,3,4,5))
  let $j := $i + 2 + 1
  return $i + $j
  
  [OUTPUT]: ( 5 , 7 , 9 , 11 , 13)
  ```

Using this framework, test files for the parser, local runtime iterators as well as Spark-enabled iterators have been added. When testing, the Spark context is initialized using the “local[*]” master parameter, meaning that it expects a local installation of Spark and it will run on the maximum number of available cores.

4.6.2 Shell and Command Line Execution Options

The project supports two ways to execute the program. From the command line, or using an interactive shell. Given the project JAR (see section 4.7 on
building/running instructions), the project can be executed on top of a Spark installation using `spark-submit`[8]. The parameters for Spark execution can be passed independently of the project, straight to the Spark framework, while the JAR expects a path to a file containing the query and a path for the output. These paths can either be on the local file-system or they can be HDFS URIs. A third optional path can be provided where the engine will write logs. An example is shown below. It passes arguments to Spark, then the project jar.

Snippet 4.18: "Using the project from the command line"

```bash
spark-submit --class jiqs.Main --master yarn-cluster --deploy-mode cluster --num-executors 50 jiqs-spark.jar --yarn-cluster
"hdfs://dco-node153.ethz.ch:8020/user/istefan/output"
"hdfs://dco-node153.ethz.ch:8020/queries/Query.iq"
```

While this is useful for benchmarks and experiments, it is not ideal for general use. To address this, a basic interactive shell based on the JLine3 framework was added. The shell can be used to write JSONiq queries and it automatically runs them and outputs the results. The shell can connect to any Spark installation including on a cluster using YARN client mode. The figure below illustrates an example:

Figure 4.6: Interactive shell usage
4.7 Build Tools, Dependencies, Licenses and Execution

4.7.1 Build

The project is built around Maven and ANT. ANT is used to invoke the ANTLR framework and generate the lexer and parser from the grammar file. The project itself can easily be built using Maven. It contains a build file supporting several build types, the most easy one compiles the code and packages all of the external dependencies into a single JAR. The project can be built using one single bash command in the project directory:

Snippet 4.19: “Project one-line build command”

maven clean compile assembly:single

4.7.2 Dependencies/Licenses

The project has several dependencies and uses several frameworks from third party developers:

- Spark 1.6.2 Libraries - Apache License
- ANTLR v4 Framework - BSD License
- org.json parser - JSON License
- JLine 3.0.2 terminal framework - BSD License
- Kryo 4.0.0 serialization framework - BSD License
Chapter 5

Experiments and Results

This chapter illustrates the experiments used to analyze the performance and behavior of this project, as well as the obtained results.

5.1 Environment, Queries and Datasets

The main testing environment was the DCO cluster provided by ETH Zurich and EPFL. The cluster has 8 nodes, each with an AMD Opteron 6212 CPU with 16 cores and 128GB of RAM. The software environment was comprised of the following installation:

- Hadoop DFS 2.7.1 and YARN
- Apache Spark 1.6.2
- Apache Ambari 2.2.0 cluster manager software

Some experiments were also run locally, on a normal notebook with an Intel Core i7-4720HQ CPU @ 2.60GHz and 16GB of RAM, running Zorba as a baseline.

For data, The Great Language Game dataset [19] was chosen. This dataset contains over 16 million objects in JSON Lines format, the file size being around 2.9GB. While this doesn’t qualify as big data, it is large enough for basic experiments. The second dataset is derived from a Reddit database of posts translated into JSON formats. The dataset is 35GB in size and contains 53 million JSON objects in JSON lines format. Finally, experiments were run on the complete Reddit database from 2008 up to 2015. The total size of this dataset is 1TB and it contains 1.6 billion objects.

The majority of the experiments used three types of queries, the first being a simple filter query with some let and where clauses, the second including an order-by clause and the last one is a grouping query. The queries are presented below (Snippet 5.1):
5. Experiments and Results

Snippet 5.1: "Main queries used for benchmarks"

```plaintext
( :WhereConfusion query: )
for $i in json-file("hdfs:///confusion-2014-03-02.json", 300)
let $guess := $i.guess
let $target := $i.target
where $guess eq $target
where $target eq "Russian"
return $i

( :OrderBy3Confusion query: )
for $i in json-file("hdfs:///confusion-2014-03-02.json", 300)
let $guess := $i.guess, $target := $i.target
where $guess eq $target
order by $target, $i.country descending, $i.date descending
return $i

( :GroupBy2Confusion query: )
for $i in json-file("hdfs:///confusion-2014-03-02.json", 300)
let $country := $i.country, $target := $i.target
group by $target, $country
return { "Language": $target,
   "Country": $country,
   "Guesses": length($i) }
```

5.2 Serialization Performance

While this project started by using the java.io.Serializable interface for any objects that need to be serialized, the very first performance improvement that was considered is to improve serialization. While java.io.Serializable is still used to serialize runtime iterators, the objects that get shuffled around (item objects, FLWOR keys, and tuples) have been shifted to using KRYO serialization which promises improved performance. Figure 5.1. shows the runtime comparison between the two versions, the first using java.io.Serializable exclusively, the second one using KRYO for items, tuples and keys (considering the 3 queries presented in section 5.1, the experiments were repeated 10 times each, average runtime is presented):

![Figure 5.1: Kryo versus java.io serialization performance](image-url)

<table>
<thead>
<tr>
<th></th>
<th>java.io Serialization - Runtime(s)</th>
<th>KRYO Serialization - Runtime(s)</th>
<th>Improvement %</th>
</tr>
</thead>
<tbody>
<tr>
<td>where</td>
<td>32.522</td>
<td>34.356</td>
<td>-5.539259578</td>
</tr>
<tr>
<td>order-by</td>
<td>100.895</td>
<td>68.621</td>
<td>31.85262426</td>
</tr>
<tr>
<td>group-by</td>
<td>145.853</td>
<td>107.658</td>
<td>26.18732559</td>
</tr>
</tbody>
</table>

The first query only filters the items and thus, no shuffling of data between nodes takes place. The data is partitioned in the first stage and no further shuffling is required. In this case, Kryo offers no performance improvements,
there is actually a small slowdown, due to the extra overhead. However, looking at the following queries which include group-by/order-by clauses and thus, a large amount of data shuffling between cluster nodes occurs, improvements between 25% and 30% are consistently observed.

5.3 Break-Even Point Analysis

The first series of tests was conducted in order to determine at which point the Spark-enabled engine is faster than a local Zorba execution. Figure 5.2 illustrates the results for the 3 types of queries presented in section 5.1, using the confusion language game dataset of around 16 million JSON objects. The complete dataset takes up around 3GB which is by no means a “big data” scenario but it is more than enough for this particular experiment.

In case of the filter query with no grouping and ordering, runtime grows linearly for the local execution, as expected. The Spark execution has some initial overhead but flattens out at under 20 seconds for this first case. At 2 million objects, it’s faster than the serial version.

Moving to the second query which sorts the objects by 3 fields, the break-even point is close to 1 million objects. The increased resource demands of sorting are observed here, since the local runtime grows at a faster rate.
5. Experiments and Results

At 16 million objects, the 16GB of RAM of the local machine are no longer enough and the runtime is thus, over 20 minutes (not plotted). Similar results can be observed in case of the third query which groups the objects by 2 fields. The memory demands of the group operation make the local runtime spike when moving from 4 to 8 million objects. The break even-point is somewhere around 2 million objects in this case, and at 16 million objects, the local execution is not usable.

This experiment proves that the project can be useful even when working on relatively small datasets.

5.4 Comparison with Elasticsearch

Elasticsearch is an open-source search engine built in Java, based on the Apache Lucene information retrieval library. It is one of the most popular full-text search frameworks at the time of this report and it is maintained by the Elastic company. Elasticsearch can run in a distributed environment and operates with the concept of "near real-time searching"[3]. Data must be "indexed" (added to a collection of documents) and afterwards, there is a small latency, usually one second, until the data can be searched and processed. Internally the data is stored and indexed in a highly optimized format and this ensures that subsequent queries are fast.

Using the second dataset (53 million JSON objects from Reddit posts) the following queries were used for JSONiq and then translated into Elasticsearch CURL based queries. Two of the queries process large amounts of data while the third one is a highly-filtering query:

Snippet 5.2: "Queries used for Elasticsearch comparison"

( : query 1 : )
for $i in json-file(“hdfs:///user/gfourny/Reddit∗”, 500)
  where $i.subreddit eq “AskReddit”
  order by $i.score descending , $i.author
  return $i

( : query 2 : )
for $i in json-file(“hdfs:///user/gfourny/Reddit∗”, 500)
  where $i.subreddit eq “AskReddit”
  let $author := $i.author, $score := $i.score
  group by $author
  let $totalScore := sum($score)
  order by $totalScore descending
  return {"Author": $author,
            "TotalScore": $totalScore}

( : query 3 : )
for $i in json-file(“hdfs:///user/gfourny/Reddit∗”, 500)
where $i\.subreddit eq \"AskReddit\" and $i\.score gt 4500
let $author := $i\.author, $score := $i\.score
group by $author
let $totalScore := sum($score)
order by $totalScore descending
return {
  \"Author\": $author,
  \"TotalScore\": $totalScore
}

Elasticsearch was deployed on the same DCO cluster where the experiments for the presented project were executed, using search version 2.4.5. Caching of query results was disabled for a more honest comparison and the results set was limited to 1000 for both the JSONiq engine and Elasticsearch. Given the nature of the queries (both contain group-by/order-by clauses), full scans would be necessary anyway. The project was tested in shell-mode, without any output being written to HDFS. The results are presented below, in figure 5.3.

Unsurprisingly, Elasticsearch is much faster than the current implementation of the JSONiq engine. The main reason is that Elasticsearch does not read actual JSON files like our engine, but instead requires them to be parsed and indexed into a specific collection. Time-wise, this is a very expensive operation that only has to be done once and all subsequent queries will be...
5. Experiments and Results

fast. Moreover, ES uses a highly optimized, compressed binary format that reduces read time. While the performance difference is large, it also provides some insight into what can be achieved in the future if enough time is spent on optimizing every aspect of the engine.

One upside of the engine is the significant improvement that this engine brings over technologies like Elasticsearch or MongoDB when it comes to ease of use and expressiveness. More complex queries that can be easily written in JSONiq become increasingly difficult to handle in Elasticsearch:

Snippet 5.3: "Equivalent query in JSONiq and Elasticsearch"

```json
for $i in json-file("hdfs://user/gfourny/Reddit", 500)
where $i.subreddit eq "AskReddit" and $i.score gt 4500
let $author := $i.author, $score := $i.score
group by $author
let $totalScore := sum($score)
order by $totalScore descending
return {
  "Author": $author,
  "TotalScore": $totalScore
}
```

```json
curl -XGET 'http://localhost:9200/redditfull2/_search?pretty' -H 'Content-Type: application/json' -d
{
  "from": 0, "size": 1000,
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "subreddit": "AskReddit"
          }
        },
        {
          "range": {
            "score": {
              "gte": "4501"
            }
          }
        }
      ]
    }
  },
  "aggs": {
    "author": {
      "terms": {
        "field": "author",
        "order": {
          "total_score": "asc"
        }
      }
    },
    "aggs": {
      "total_score": {
```
5.5 Speedup Analysis

Speedup analysis experiments were also executed in order to observe the behavior of the engine in more detail. This also done in order to determine the optimal number of executors for each dataset individually. Figure 5.4. illustrates the runtime and the corresponding speedups on the Language Game dataset. For this particular dataset which is only 2.9GB in size, the optimal number of executors is around 45. Above this number and the shuffling overhead begins to outweigh the benefits of the extra compute power. Overall, the engine has good scalability.

![Figure 5.4: Performance Analysis on Language Game Dataset](image)

5.6 Processing Large Datasets

In order to see how the engine behaves in “big data” related scenarios, the final set of tests experimented with the full Reddit dataset from 2007 to 2015. Around 1.6 billion JSON objects with a total size of around 1TB. The dataset was further duplicated up to 4 times. The cluster was expanded for this experiment, moving up from 8 nodes to 64 nodes. On the other hand, in order to handle this large amount of data, the cluster was switched from SSDs to larger but slower standard HDDs. The results of a filter query and a grouping query are presented in Figure 5.5. The system is stable when
5. Experiments and Results

dealing with large amounts of data even though the response time is not quite linear.

Figure 5.5: Performance Analysis on the full Reddit dataset

For the grouping query, the first straggler executors began to appear at 4.8 billion objects. While most executors finished the final stage in under 5 minutes, one executor dragged on for over an hour. Increasing the partition number from 3000 to 15000 and enabling Spark’s speculative execution mode (which restarts slow tasks) solved the issue. Further increases of the dataset size resulted in frequent executor crashes indicating that a limit was reached for this particular setup.
Chapter 6

Conclusions and Future Work

At the time of this report, the engine only partially supports the features of JSONiq. Implementing the missing features (see the Appendix) would be a natural next step for this project. Some of the features are straightforward to implement while other may require more development hours. Some major features yet to be added include:

- **JSONiq Modules.** Modules allow grouping of functions and variables into distinct entities.

- **Count clauses and positional variables.** As explained in section 4.5.3, implementing JSONiq for-clause positional variables in Spark is problematic because there is no support for read/write, globally synchronized variables. Implementing this would require extra research in that direction.

- **try-catch Blocks - Exception handling.** The addition of exception handling mechanisms would further improve the usability of the query engine.

- **XQuery error codes and W3C test suite** In order to make this project ready for a potential release, the engine is required to throw appropriate error codes that match those in the XQuery specification. Extending the testing framework to incorporate the massive number of tests from the W3C test suite would help with bug-fixing.

- **A better CLI and extensive documentation** The final step would be to document the engine and improve the CLI.

Research concerning the project could also be pushed in new directions by trying to enable support for FLWOR window clauses on top of a streaming engine like Apache Flink, or improve the validation aspect by adding support for schema validation (JSON Schema or JSound).
The results prove the feasibility of the design and its potential to achieve impressive performance. There are many optimizations that can improve the performance of this engine, some of the more immediate ones being:

- **Using a custom JSON parser.** Right now in order to parse JSON, the engine is using the org.json parser to convert the input from strings to a representation of JSON objects and then another step is taken to convert from this internal representation to instances of item types. This process could be reduced to one single stage with a custom parser.

- **Extending KRYO serialization to all runtime iterators.** While Kryo is used for items, FLWOR tuples and sort/group keys, replacing the java.io serialization of the runtime iterator instances (setup time) will improve runtime performance by a constant factor.

- **Improved architecture for certain Runtime Iterators** Improving the architecture of the runtime iterators (mostly in the local runtime iterators) by increasing the streaming degree would improve performance overall.

The project can be extended in other directions as well. Since the file reading functions are essentially a wrapper over Spark’s **textFile** function, any storage system supported by Spark including local file-systems, HDFS, Amazon S3 are supported in this engine as well. The engine could be extended to run on top of Elasticsearch as well.

This project presented the idea of a new JSONiq engine that can run on top of Spark. Theoretical aspects of mapping FLWOR clauses to Spark transformations and tuple streams to RDD, as well as providing a partial implementation that was successfully tested on a cluster. In addition to implementing the missing features, the Elasticsearch comparison proves that there is a lot of room for improvement.

The final step would be to release this project as open-source software (“Sparksoniq” could be the name) under the Apache license.
### A.1 Implemented Features

This appendix contains a table with all of the JSONiq features and the current state of this project feature-wise.

<table>
<thead>
<tr>
<th>JSONiq Feature</th>
<th>Implemented</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntegerLiteral</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>DecimalLiteral</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>StringLiteral</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>BooleanLiteral</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>DoubleLiteral</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>NullLiteral</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Object constructor</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Array constructor</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Unquoted keys</td>
<td>NO</td>
<td>-</td>
</tr>
<tr>
<td>Merging object</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>constructor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comma</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Range</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Parenthesis</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Addition</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Subtraction</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Division</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Integer division</td>
<td>YES</td>
<td>-</td>
</tr>
<tr>
<td>Modulo</td>
<td>YES</td>
<td>-</td>
</tr>
</tbody>
</table>
### A. Appendix

<table>
<thead>
<tr>
<th>JSONiq Feature</th>
<th>Implemented</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>String concatenation</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Value comparison</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>General comparison</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Or</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Not</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>And</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Universal quantifiers</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Existential quantifiers</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Object field selector (quoted string)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Object field selector (unquoted string)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Object field selector (nested)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Object field selector (variable)</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Object field selector (context item)</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Array member selector</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Array unboxing</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Sequence filter</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>Context item</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>If then else</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>switch</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>try catch</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>let</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>order by</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>count</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>group by</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>window</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>instance of</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>treat as</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>cast</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>typeswitch</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>prologs</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>modules</td>
<td>NO</td>
<td></td>
</tr>
<tr>
<td>Function library</td>
<td>YES</td>
<td>Implemented functions: max, min, count, sum, avg, keys, values, size</td>
</tr>
</tbody>
</table>


Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor’s thesis, Master’s thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):
JSONIQ ON SPARK

Authored by (in block letters):
IRIMESCU
STEFAN

For papers written by groups the names of all authors are required.

Name(s):
IRIMESCU

First name(s):
STEFAN

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the ‘Citation etiquette’ information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
ZURICH, 26.08.2017

Signature(s)
IRIMESCU STEFAN

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.