



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich



Master's Thesis Nr. 327

Systems Group, Department of Computer Science, ETH Zurich

A Test Suite for Rumble

by

Stevan Mihajlovic

Supervised by

Dr. Ghislain Fourny, Prof. Dr. Gustavo Alonso

October 1, 2020 - April 1, 2021

DINFK

Abstract

Insert some short text here

Acknowledgements

Write some sad story and then a positive story. Don't forget to mention all the people

Contents

Contents	v
1 Introduction	1
2 Background and Related work	3
2.1 Big Data	3
2.2 Hadoop	5
2.2.1 HDFS	5
2.2.2 MapReduce	6
2.3 JSONiq	7
2.4 Spark	7
2.5 JSONiq	7
2.6 Rumble	8
3 XQuery/XPath 3.* Test Suite(QT3TS)	9
3.1 Analysis	9
3.1.1 Programming Language	9
3.1.2 Data Format	9
3.1.3 XML Parser	10
3.2 Phase 1 Implementation	11
3.2.1 Description	11
3.2.2 Architecture	13
3.2.3 Results	13
3.3 Phase 2 Implementation	14
3.3.1 Description	14
3.3.2 Architecture	18
3.3.3 Results	19
3.4 Phase 3 implementation	20
3.4.1 Description	20
3.4.2 Architecture	23

CONTENTS

4 Concluding Remarks	25
4.1 Overall Summary	25
4.2 Open Problems	25
Bibliography	27

Chapter 1

Introduction

The increasing amount of data available to process, as well as the ever-growing discrepancy between storage capacity, throughput and latency, has forced the database community to come up with new querying paradigms in the last two decades. Data became nested and heterogeneous (JSON), and is increasingly processed in parallel (Spark). In order to make querying more efficient and accessible, Rumble [MFI⁺20] is an engine that automatically runs queries on semi-structured and unstructured documents on top of Spark, using the JSONiq language.

JSONiq [jso20] is a functional and declarative language that addresses these problems with its most useful FLWOR expression which is the more flexible counterpart of SQL's SELECT FROM WHERE. It inherits 95% of its features from XQuery, a W3C standard.

The XQuery/XPath 3.* Test Suite (QT3TS) [W3C13] provides a set of tests with over 30000 test cases designed to demonstrate the interoperability of W3C XML Query Language, version 3.0 and W3C XML Path Language implementations.

The high level idea of this work is to implement a Test Driver that can directly use QT3TS in order to test and verify Rumble implementation.

Chapter 2

Background and Related work

In this chapter, we will introduce context on which our work is based. For full overview, we must familiarize the reader with the following concepts: Big Data, NoSQL, MapReduce, YARN, Spark, JSON, JSONiq and finally Rumble.

Test Driver itself will be built as a layer on top of Rumble. Because of the architecture which enables data independence, we do not need to know its under-laying structure. However, seeing the full architecture and having an overview will help us make decisions throughout this work.

2.1 Big Data

Big Data in today's world has a broad scope and several definitions. Here we will present a certain view of the Big Data on which Rumble was based. We can look at the data being "big" in following 3 dimensions:

- Volume - These term simply corresponds to the amount of bytes that our data consists of. To have idea of the scale, in Big Data we are often looking at PB of data. Scientific centers such as CERN produce tens of PB of data annually. The information, the data in today's world brings value. Not only scientific centers, but also big companies gather data, store it in their data centers and process it in order to extract this value.
- Variety - Data often comes in different shapes. The most familiar ones are Text - completely unstructured data, followed by data organized as Cubes, Tables, Graphs or Trees on which we will mainly focus. Until 2000's, the world was mainly oriented towards Relational Databases for which they under-laying shape is Table. Main focus was on introducing normalization forms with the idea to avoid data redundancy. Then the tables would simply be joined using SQL as the query language via the foreign keys. However, starting from 2000's Relational Databases and SQL could not satisfy the needs of real world scenarios. Often data is

can I quote Fourny's lecture

2. BACKGROUND AND RELATED WORK

unstructured, nested, values are missing etc. This trend led to NoSQL Databases. Main focus in NoSQL Database is to perform opposite and actually de-normalize the data. Looking at the table, we would now allow non-atomic values in a single cell or even missing values. Such a transition leads the data shape to transform from flat homogeneous tables to nested heterogeneous trees. Choosing the correct data shape is essential. What CSV and SQL were in relational database, for tree shaped data we have JSON and XML as a data format with JSONiq and XQuery as their respective querying languages.

- Velocity - Data in the end is physically stored on some medium drive. The 3 main factors of this under-laying medium drive are Capacity, Throughput and Latency. From mid 1950's until today we have witnessed tremendous increase in all 3 factors. Capacity has increased by up to 200×10^9 , throughput by 10×10^3 and latency by 200 times. This ever-growing discrepancy between factors has brought needs for parallelization and batch processing. Since a single medium drive has increased capacity much more compared to throughput we need to read data from multiple medium drives at the same time in parallel to be able to obtain data fast enough. At the same time, to face discrepancy between throughput and latency, we need to obtain data in batches. Thus, the need for systems that can perform parallel batch processing has increased.

maybe insert a picture from presentation and also maybe link to some work

In summary, traditional RDBMS such as Oracle Database or Microsoft SQL Server, have focused on being compliant with ACID (Atomicity, Consistency, Isolation and Durability) properties. Such RDBMS with homogeneous tables are good when handling small amount of data. However, if we need to massively scale the data, we need to turn to different technologies. These traditional RDBMS that use File Systems such as FAT32 or NTFS for physical storage are not sufficient.

maybe quote the second presentation

NoSQL (Not Only SQL) Databases on the other hand are compliant with CAP (Capability, Availability, Partition tolerance) theorem. Examples of new NoSQL databases that have emerged are key-value stores (DynamoDB), column-oriented stores (HBase) and document stores (MongoDB). They often use Distributed File System as physical storage such as HDFS. Instead of traditional scaling up, by buying single high-performance hardware, the orientation is towards scaling out by buying a lot of cheap commodity hardware. Such scaling enables that hardware costs grow linearly with the amount of data. All these concepts lead to building high performance and scale-able frameworks such as Hadoop that can query and process distributed massive data in parallel.

maybe quote third presentation

2.2 Hadoop

Apache Hadoop is an open source framework written in Java that is able to manage big data, store it and process it in parallel in a distributed way across cluster of commodity hardware. It consists of 3 components:

- HDFS - Storage layer
- MapReduce - Processing layer
- YARN - Resource management layer

In this section we will briefly introduce each of the layers. It will help reader to better understand Spark in the upcoming chapter.

2.2.1 HDFS

Hadoop Distributed File System - HDFS is a physical storage layer of Hadoop inspired by Google File System written in Java. It is one of the most reliable FS for storing big data distributed on a cluster of commodity hardware.

In this section, we need to understand how HDFS physically stores big data on the machines. When we say big data, we are thinking at scale of millions of PB files. This means that files are bigger than single drive (medium). Therefore, in such a setting, the most suitable is block storage. Unlike typical NTFS system with allocation units of 4 KB, the block size is by default 64 or 128 MB. It is chosen as a good trade off between latency and replication. Transferring multiple blocks bigger than 4 KB will reduce latency and also reduce the network overhead. When it comes to replicas, each block has by default 3 copies in case of a failure.

The architecture is master-slave. Master is NameNode and it is in charge of storing the namespace. Namespace is hierarchy of files and directories. Since the blocks are 64 or 128 MB, the metadata is also small. And since we are storing rather small amount of very large files, the whole namespace can fit in RAM of the NameNode. In addition to namespace, NameNode knows the file to block it consists of mapping together with location of block and its replicas. The blocks are stored on DataNodes that act as slaves. When clients want to read/write the files they communicate with NameNode only once to receive the locations meaning that NameNode is not the bottleneck.

Such an architecture allows potential infinite scalability just by adding DataNodes meaning that hardware cost grows linearly with the increase of data. The single point of failure is NameNode, meaning that from CAP theorem we have Capability and Partition tolerance at the cost of Availability. In case of a failure, there is a secondary NameNode that would start-up. Also it enables high durability with 3 replicas and read/write performance. When reading, it usually transfers a lot of data - batch processing.

2.2.2 MapReduce

MapReduce in most broad definition is a programming model (style). It answers to the question how we process the data and it consists of two crucial steps map and reduce alongside with shuffle as the intermediate step:

- Map - Input data is mapped into intermediate set of key-value pairs
- Shuffle - All key-value pairs are shuffled in a way such that all pairs with the same key end up same machine
- Reduce - Data is aggregated on the machine and the output is produced

Maybe insert picture

Example - Count occurrences of each word in a document of 1000 Pages:

1. What we can do is that we can first have a single map task per a page. This way those 1000 pages can be done in parallel. The map task will perform $\text{Map}(K1, V1) \rightarrow \text{List}(K2, V2)$, where $K1$ is in range from 1 to 1000 (for each page) and $V1$ is the text on each page. $K2$ will have values in range of all possible words that occur in the document. $V2$ will always be 1. Such a mapper is very primitive. In case that reduce task is a function that is commutative and associative then it is allowed to execute same function in map task to reduce the amount of shuffle that will happen afterwards. As count is such a function, in map task we can already perform sum per key. It means that $K2$ stays same and $V2$ will be the actual count per page!
2. As not all possible words will appear in all pages, we will simply put together collection of all the produced key value pairs and sort them per key. We will then assign all key-value pairs with the same key to the single reducer and partition the data accordingly.
3. Reduce task will perform $(K2, \text{List}(V2)) \rightarrow \text{List}(K2, V3)$ - Reducer can output the same key value pair, but in general it can be any other. Finally $V3$ will be the sum of occurrences of the word $K2$!

In general MapReduce as programming model can be used in any framework with any under-laying physical storage such as Local File System, S3, Azure, HDFS. Here we will describe infrastructure in Hadoop version 1 where it is running on top of HDFS where we also have Resource Management layer. It is again master-slave architecture where we have JobTracker and TaskTracker. JobTracker is the master with responsibilities of Resource Management, Scheduling, Monitoring, Job lifecycle and Fault-tolerance. One Job contains from multiple tasks, depending on how the data is split. And 1 task can be map or reduce task. 1 or more tasks are then assigned to TaskTracker that need to execute them. JobTracker is collocated with NameNode and TaskTracker usually with the DataNode in order to bring query to the data.

2.3 JSONiq

JSON as data format supports 6 types: Object that act as maps from string to any other value, Arrays used to nested that can contain any sequence, Strings, Numbers, Boolean and null.

2.4 Spark

Spark is generalization of Map Reduce - programming model that consists of two steps Map and Reduce. It generalizes it to DAGs that are built around RDDs which are partitioned collection of values. DAG is basically a physical plan of execution. A DAG is created when the user creates a RDD (by reading from file for example) and applies chains of lazy transformations on it. When the action is called it triggers computation. The DAG is given to the DAG Scheduler which divides it into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The Stages are passed on to YARN that now executes it physically. Since Spark has end to end DAG, it can figure out which tasks can be done in parallel. All these will then run into parallel on several nodes. When it comes to data processing Spark supports both batch and real time processing and with that is directly able to deal with discrepancy between capacity, throughput and latency.

DataFrame is high level abstraction of RDD's. It is logical data model that enables users to view and manipulate data independently of physical storage. DataFrames store data in collection of rows enabling user to look at RDD's as tables. They are nothing more than named columns like we had before. Therefore, we can use high level declarative language - Spark SQL to query the data without caring about under-laying physical storage.

The main problem with DataFrames is that heterogeneous data that we are encountering in tree data shapes cannot fit in DataFrame. All the de-normalization that enabled nested, missing values or values of different type will not work. Running Spark on such a DataSet results in Spark skipping and leaving to user to manually handle heterogeneous data. DataFrames are simply not the correct representation for the tree shaped data.

2.5 JSONiq

JSONiq as mentioned earlier is a querying language designed to analyze tree shaped data. It has data model that is able to capture all aspects of JSON data format. All 6 types that JSON supports are represented using Sequence of Items. Then all Expressions that exist operate only on Sequence of Items. The main observation is that Expression takes Sequence of Items as the input and as the output produces again Sequence of Items. This means that expres-

2. BACKGROUND AND RELATED WORK

sions can be nested in any desired way. The Expression can be Arithmetic, Logic, Comparison, Literal, JSON construction, JSON navigation, Sequence Construction, Built-in function or a FLWOR expression. FLWOR expression is the most powerful one and it is equivalent to Select From Where in SQL with an addition that it can be nested any number of times in almost any order.

2.6 Rumble

include picture from the presentation with Layers of JSONiq and Spark, also the picture with AST architecture

Chapter 3

XQuery/XPath 3.* Test Suite(QT3TS)

3.1 Analysis

In this chapter, we will discuss design decisions that we have made during the development of Test Driver. The core idea is to develop Test Driver completely independently from Rumble by maintaining the code outside of Rumble.

3.1.1 Programming Language

We view Rumble as black-box and the single point of communication with Rumble should only be via the Rumble Java public API. Therefore, we have decided to implement Test Driver as Java Console Application. Furthermore, Rumble is also written in Java. We decided to setup our Java Console Application project to have two modules - Test Driver and Rumble module. Rumble module is the branch in repository created for the purpose of this work [Mih20]. Making Test Driver module dependent on it, we are allowing possibility to directly use Rumble and its classes in case that not everything is possible to be achieved by treating Rumble as the black-box.

3.1.2 Data Format

The XQuery/XPath 3.* Test Suite (QT3TS) is publicly available at W3C Public CVS Repository under module name 2011/QT3-test-suite [W3C11]. Since April 1st 2019, CVS tree has been discontinued and the repository has been migrated to W3C Public GitHub repository [W3C20]. The tests are published as a set of files - test sets containing in total more than 30000 test cases. The tests are published as a set of files, mostly in XML format. W3C does not supply a Test Driver for executing the tests. Instead, for each implementation a Test Driver should be written. As these test sets are mostly written in XML format, the first component that our Test Driver will require is the XML parser.

3.1.3 XML Parser

XML parser is a program that allows our application to read and write XML documents. For our work, we have investigated following possibilities:

- DOM (Document Object Model) - This parser loads entire XML document in memory and uses interfaces to access the information. It can access couple of item elements at same time. It can be used for both reading and writing.
- SAX (Simple API for XML parsing) - This parser doesn't load XML document in memory. Instead, it allows us to register a handler with SAX parser. When parser goes through file it keeps invoking methods on the handler class for each item. It process it in sequence one at a time. For each new item it reads, it forgets state of previous items. Therefore, on each read we need to take appropriate action in our application. It is read only and also known as push parser. There is no handler on XML document side, only in our application.
- STAX (Streaming API for XML parsing) - This parser allows us to both read and write multiple documents at same time. Unlike SAX that reads one item at a time, STAX can be explicitly asked to get a certain item from XML document without loading it in memory. Therefore, we can look at it as mixture of DOM and SAX. It is pull parser and has handler on XML document as well
- JAXP (JAVA API for XML parsing) - Since JDK 1.5, the JAXP API has been available as a standard part of the Java platform, and it provides access to XSLT transformation, schema validation, and XPath processing services.
- Saxon [Kay20] - Open Source XSLT & XQuery processor developed by Saxonica Limited. The Saxon package is a collection of tools for processing XML documents. The main components accessible via API are:
 1. XSLT processor. Saxon implements the XSLT 3.0 Recommendation. The product can also be used to run XSLT 2.0 stylesheets, or XSLT 1.0 stylesheets in backwards compatibility mode.
 2. XPath processor. This supports XPath 2.0 and XPath 3.1. It can also be used in backwards-compatibility mode to evaluate XPath 1.0 expressions.
 3. XQuery processor. This supports XQuery 3.1, which also allows XQuery 1.0 or 3.0 queries to be executed.
 4. XML Schema Processor. This supports both XSD 1.0 and XSD 1.1. It can be used to support the schema-aware functionality of the XSLT and XQuery processors.

For parsing XML, we have decided to use Saxon. One may argue that for all 4 listed components, Java also has its own API – JAXP for 1st, 2nd and 4th together with XQJ for 3rd. However, in practice, Saxon is easier to use and more flexible than JAXP. Apart from that, main arguments are:

1. Saxon itself is one of the implementations for which Test Driver was also implemented. Based on Results Report [Kay16], it passes more than 99,9% of the QT3TS tests.
2. Saxons implementation of the Test Driver can be used as a baseline for developing our own Test Driver.

3.2 Phase 1 Implementation

3.2.1 Description

In the first phase of the implementation we have analyzed the structure of QT3TS. We had to understand the under-laying structure of each and every test case. We had to see under which tags the information is stored in order to obtain it using Saxon API. Example test case in XML format:

```
<test-case name="fn-absint1args-1">
  <description>
    Test: absint1args-1 The "abs" function
    with the arguments set as follows:
    $arg = xs:int(lower bound)
  </description>
  <created by="Carmelo Montanez" on="2004-12-13"/>
  <environment ref="empty"/>
  <test>fn:abs(xs:int("-2147483648"))</test>
  <result>
    <all-of>
      <assert-eq>2147483648</assert-eq>
      <assert-type>xs:integer</assert-type>
    </all-of>
  </result>
</test-case>
```

The two most important tags in each test case are:

- Test - this is the test that should be executed on Rumble. It can be XSLT, XPath or XQuery expression.
- Result - this is the expected result outcome of the test tag. As it can be seen in the provided example, there are several types of assertions that we need to verify.

3. XQUERY/XPATH 3.* TEST SUITE(QT3TS)

Test Driver's Test Case Handling Logic is supposed to iterate over catalog.xml using the Saxon API. This XML document contains list of all test-sets. Again, using the Saxon API, we iterate over test-cases in each of the test-sets. For each test-case, we are asking explicitly Saxon XML parser to get items under Test and Result tags. To use Saxon API, we need to know the structure. But, once Test Case Handling Logic obtains information under Test tag, it passes it down "as is" to Rumble API in order to execute the query. Rumble API returns the result which is then passed down to Test Result Handling Logic.

Test Driver's Test Result Handling Logic is in charge of determining which assertion needs to be performed. Here we provide the list of possible assertions:

- assert-empty - This assertion requires result to return empty sequence
- assert - This assertion requires us to run another query in which obtained result will be used as parameter of the new query. For example:

```
<test>math:acos(0)</test>
<result>
  <assert>
    abs($result - 1.5707963267948966e0) lt 1e-14
  </assert>
</result>
```
- assert-eq - It requires us to run another query in form of obtained result "eq" value under the assert-eq tag
- assert-deep-eq - Similar to assert-eq but runs "deep-equal" query
- assert-true - It requires result to return single Boolean value True
- assert-false - Opposite of assert-true
- assert-string-value - It requires that each item in the obtained result sequence is type of String and also "eq" to the sequence under this tag
- all-of - It contains multiple different assert tags described in this list and it requires all of them to be fulfilled
- any-of - Similar to all-of but requires only one of them to be fulfilled
- assert-type - Requires to check if obtained result is instance of this tag
- assert-count - It requires obtained result sequence size to be equal to the value under assert-count tag
- not - It requires to execute nested assertion with a negation

After the assertion is performed, we need to classify the results. The idea is to make statistics that are described in 3.2.3. With such a classification we would be able to improve Rumble by reporting bugs in its implementation.

3.2.2 Architecture

The overview of scenario described in 3.2.1 can be seen in Figure 3.1

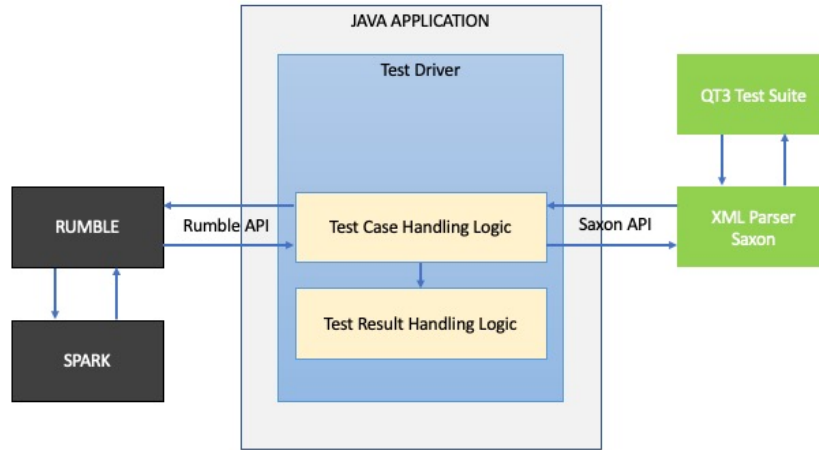


Figure 3.1: Phase 1 Architecture Overview

3.2.3 Results

As explained in 3.2.1, result obtained via Rumble API was compared with the expected result by applying the correct assertion check. In case assertion passed, test-case was considered a Success and otherwise Fail. The block of code performing these operations was surrounded by try and catch. In case that test failed because the syntax was not completely JSONiq, it would throw a RumbleException or more generally an Exception - Crash. With this implementation, we were to be able to distinguish 3 possible scenarios:

1. Success - Test case succeed
2. Fail - Test case failed because of bug in Rumble
3. Crash - Test case failed because it is not compatible with Rumble

The report is generated as .csv file having test-sets as rows and total number of test-cases per scenario in the columns. In Table 3.1 we will present the aggregated sum over all rows in the .csv file:

Scenario	Total test-cases	% of all test-cases
Success	2330	7.8
Fail	2769	8.8
Crash	26421	83.4

Table 3.1: Phase 1 Results Overview

3.3 Phase 2 Implementation

3.3.1 Description

After generating Phase 1 Implementation report described in Table 3.1, we carefully examined our implementation and identified 4 major pain points:

- Unstable implementation of assertion which resulted in implementing proper way of result binding in Rumble
- Too many crashing tests which resulted in implementing converter
- Insufficient granularity for distinguishing test-cases
- Improving Test Driver implementation resulted in breaking previously implemented features. Therefore, regression tests were introduced

Result Binding

To better understand the issues we have encountered, we will provide following code snippet:

```
private boolean AssertEq(List<Item> resultAsList,
    XdmNode assertion) throws UnsupportedOperationException {
    String assertExpression = assertion.getStringValue();
    List<String> lines = resultAsList.stream()
        .map(x -> x.serialize()).collect(Collectors.toList());
    assertExpression += "=" + lines.get(0);
    List<Item> nestedResult = runQuery(assertExpression);
    return AssertTrue(nestedResult);
}

private boolean AssertStringValue(List<Item> resultAsList,
    XdmNode assertion) throws UnsupportedOperationException{
    String assertExpression = assertion.getStringValue();
    List<String> lines = resultAsList.stream()
        .map(x -> x.serialize()).collect(Collectors.toList());
    return assertExpression.equals(String.join(" ",lines));
}
```

If we examine the AssertEq implementation, we will notice that lines.get(0) assumes that obtained result is a single item and takes first one. It does not handle sequences! Furthermore, handling sequences was only possible for AssertStringEqual in case that our result is sequence of strings by performing string concatenation. All other assertions such as Assert, AssertEq, AssertDeepEq are not possible to be implemented. Finally, if we remember Assert example from Section 3.2.1, we will notice that we had to perform string replace of \$result with actual result obtained from the Rumble API.

Thus, Rumble was extended to support result binding. The change was made in Rumble implementation itself. The only modification it required was to instantiate a new RumbleConfiguration and also new Rumble instance for each test-case that requires result binding. Check code snippet below:

```
private boolean Assert(List<Item> resultAsList,
    XdmNode assertion) throws UnsupportedOperationException {
    String expectedResult = Convert(assertion.getStringValue());
    return runNestedQuery(resultAsList, expectedResult);
}

private boolean runNestedQuery(List<Item> resultAsList, String expectedResult,
    RumbleRuntimeConfiguration configuration = new RumbleRuntimeConfiguration(),
    configuration.setExternalVariableValue(
        Name.createVariableInNoNamespace("result"),
        resultAsList);
    String assertExpression = "declare variable $result external;" + expectedResult;
    Rumble rumbleInstance = new Rumble(configuration);
    List<Item> nestedResult = runQuery(assertExpression, rumbleInstance);
    return AssertTrue(nestedResult);
}
```

The main concern of the new implementation was that performing many instantiations might cause the execution time to increase dramatically. However, after run-time increased only by 15seconds from 2minutes - only 12.5%.

Once the result binding was implemented, it allowed us to run the assert type also as a query instead of calling the publicly exposed methods of the Item class in the Rumble Java API. In Phase 1 implementation, we had a switch case for every possible type that Rumble Java API supports, making code difficult for future maintenance and extension with new supported types. With running assert type as "instance of" query, we managed to have a single point of conversion performed in the beginning and applied for both test case and the expected result. Within conversion we would discover the unsupported type errors without the need of second switch case to check whether Rumble's API Item class supports the type or not. Furthermore, the previously implemented switch case had unsupported type as default therefore hiding some types that were supported but not specified in the documentation. The mentioned conversion will be explained more in detail in Section 3.3.1.

The clean separation that was performed here initialized idea and was a base plan for XQuery to JSONiq conversion logic separation. In Section 3.4.1 we will describe Architecture that has separate application that takes XQuery as input, performs conversion and outputs JSONiq test suite. Such approach would make the Test Driver easily maintainable and extensible!

Converter

As seen in Table 3.1, we had less than 10% Success test-cases as almost all of them required conversion to JSONiq. Here we will document all the conversions that we have performed on both Test and Result tags in this Phase.

The first conversion that we have performed is between types. Both XQuery and JSONiq have simple(atomic) and complex(non-atomic) types.

The list of atomic types that is currently supported by Rumble was taken from official Rumble documentation [IFM⁺20b] and conversion was implemented accordingly. For all types that are not supported, our code throws `UnsupportedTypeException`.

Following 3 complex (non-atomic) types were handled by following conversion:

1. `array(*)` was replaced with `array*`
2. `item()` was replaced with `item`
3. `map(string, atomic)` was replaced with `object`

On the other hand, following 7 complex (non-atomic) types could not be converted and they all throw `UnsupportedTypeException`:

1. `document`
2. `element`
3. `attribute`
4. `text`
5. `comment`
6. `processing-instruction`
7. `xs:QName`

Other conversions that were performed:

1. `true()` was replaced with `true`
2. `false()` was replaced with `false`
3. `INF` was replaced with `Infinity`
4. array access via `.` was replaced with `$$`
5. `'` was replaced with `"`
6. prefixes `fn`, `math`, `map`, `array` were removed

Other items that were unsupported in Phase 2 were `node()`, `empty-sequence()` and `xs:NOTATION` together with all error codes that are not in Table 3.2 that was taken from [IFM⁺20a].

Type	Status	Supported Error Codes
atomic	supported	FOAR0001
anyURI	supported	FOCA0002
base64Binary	supported	FODC0002
boolean	supported	FOFD1340
byte	not supported	FOFD1350
date	supported	JNDY0003
dateTime	supported	JNTY0004
dateTimeStamp	not supported	JNTY0024
dayTimeDuration	supported	JNTY0018
decimal	supported	RBDY0005
double	supported	RBML0001
duration	supported	RBML0002
float	not supported	RBML0003
gDay	not supported	RBML0004
gMonth	not supported	RBML0005
gYear	not supported	RBST0001
gYearMonth	not supported	RBST0002
hexBinary	supported	RBST0003
int	not supported	RBST0004
integer	supported	SENR0001
long	not supported	XPDY0002
negativeInteger	not supported	XPDY0050
nonPositiveInteger	not supported	XPDY0130
nonNegativeInteger	not supported	XPST0003
positiveInteger	not supported	XPST0008
short	not supported	XPST0017
string	supported	XPST0080
time	supported	XPST0081
unsignedByte	not supported	XPTY0004
unsignedInt	not supported	XQDY0054
unsignedLong	not supported	XQST0016
unsignedShort	not supported	XQST0031
yearMonthDuration	supported	XQST0033
		XQST0034
		XQST0038
		XQST0039
		XQST0047
		XQST0048
		XQST0049
		XQST0052
		XQST0059
		XQST0069
		XQST0088
		XQST0089
		XQST0094

Table 3.2: Rumble Supported Types and Error Codes

Regression Tests

During Phase 1, we were performing iterations with goal to overall improve Test Driver's implementation. The good metric while performing these iterations was total number of test-case Crashes. Our goal was to reduce those numbers as much as possible. This was mainly handled by making following changes: bug fixes, software enhancements, configuration changes. Creating this changes in software development can usually lead to creating new issues that were not present before or re-emergence of old issues. In these cases it is quite common that software development requires regression testing. Regression testing (rarely non-regression testing[1]) is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change.[2] If not, that would be called a regression. . During iterations, it was noticed that our approach of fixing and improving the application is highly exposed to changes that require regression testing.

While performing iterations, we had to ensure that any further implementation would not break the test-cases that were passing before and at the same time not introduce new test-cases that are Crashing. Thus, for each iteration we have maintained log files of all Passed (Success + Managed) and Crashed test-cases. In every next iteration we have done two comparison between new and previous log files. We have performed a check that compared whether all the passed test-cases from the previous implementation were also contained in the new implementation or not and created "List of test cases that were passing before but not anymore". For Crashes, we did opposite check and created list of "Tests that were not crashing before, but are now and not in list above".

3.3.2 Architecture

The overview of scenario described in 3.3.1 can be seen in Figure 3.2

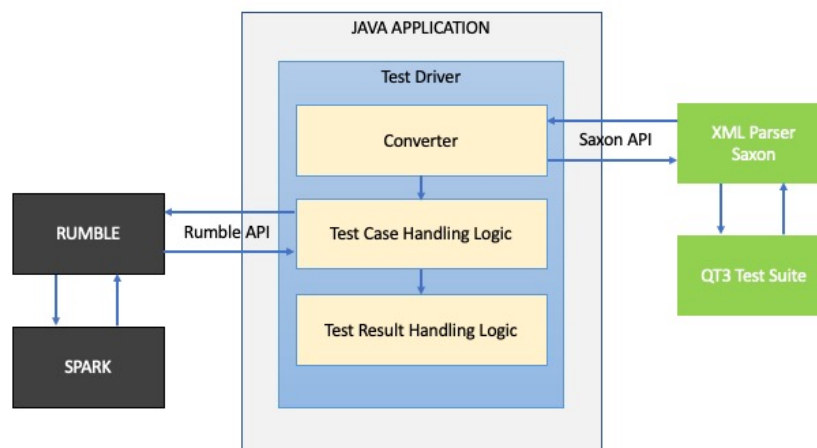


Figure 3.2: Phase 1 Architecture Overview

3.3.3 Results

As we have seen in Section 3.3.1, Crashes were not only capturing tests that are not JSONiq and needed conversion. They were also including the tests that could not succeed simply because Rumble does not support that feature, type or error code. In Table 3.2 we can see the limited amount of supported types and error codes compared to XQuery w3 specification for which we are able to verify assertion. All others would then be ignored and classified differently. Furthermore, some of them were introducing dependencies. For example, in dependency tag it was possible to have request for particular version of XPath, XQuery or XSLT. While Rumble is backwards compatible with all versions of XPath and XQuery, it does not support XSLT. We have therefore created and divided test cases into 7 groups:

1. Success – Test that is passing the assertion and does not need Converter
2. Managed – Tests that would have failed assertion, but they were modified with hard-coded conversion into JSONiq using Converter
3. Skipped – Test that is not JSONiq and thus expectedly fails assertion. These tests should not be converted to JSONiq
4. Failed – Tests that are failing because there is a bug in Rumble API or Test Driver implementation.
5. Dependency – Tests that are failing because dependency is not supported
6. Unsupported – Tests that are failing because type, feature or error code is not supported yet
7. Crash – Any other exception

After introduction of the 7 above mentioned cases, together with small adjustments and bug changes, we were able to obtain:

Scenario	Total test-cases	% of all test-cases
Success	2686	8.52
Managed	4211	13.36
Fail	2554	8.10
Skipped	5	0.02
Dependency	1481	4.70
Crash	13171	41.79
Unsupported	7412	23.52

Table 3.3: Phase 2 Results Overview

Managed category was introduced as it was identified that with simple hard-coded conversion we are able to obtain around 4200 passed tests increasing

total percentage of passed tests by roughly 12%. At first, it seems that Success and Managed should be grouped into single category, but we decided to keep them separated. The reason behind is that while fixing bugs in both Rumble and Test Driver, we will increase the number of Success test cases. At the same time, we want to keep the track of Managed ones, because in Phase 3 Implementation we are planning to generalize the hard-coded conversion and create pure JSONiq Test Suite based on given XML ones.

For Skipped tests, these are the ones that it would not make sense to try to convert them to JSONiq. One example is XSLT tests and those should be skipped. We are keeping them in separate list as in Phase 3 Implementation we will skip from output and not include them in pure JSONiq Test Suite.

The main goal of performing iterations was to go through all the crashes and try to completely eliminate them. By doing so we would also improve the statistics by classifying them into other categories. At the same time, we were manually investigating test cases and trying to find the root cause. For some of them, our Test Driver implementation was improved. For some it was identified that the XQuery function was not yet supported by Rumble or it was having bugs so Rumble implementation was also improved. List of dependencies that were found in Test Suite were documented and classified according to Rumble documentation. The list is presented in Table 3.4 .

maybe some reference here, ask Fourny

Final goal was to identify test-cases that fail but can be converted to JSONiq. They could not be included into the automatic distinction of 7 above mentioned cases and had to be handled manually. They also helped with identifying what Phase 3 conversion also had to support.

3.4 Phase 3 implementation

3.4.1 Description

The main issue of Converter described in Section 3.3.1 was that it was hard-coded conversion using Java String.replace method. Such implementation can be very unstable. For example, we can look at 5th item of "other conversions" mentioned in Section 3.3.1 - replacing ' with ". For example, test-case Literals009 is verifying whether "test' is a valid String Literal. With our hard-coded conversion, we will make this test-case valid String Literal instead of it causing an Error Code XPST0003. Therefore, we have decided to implement Test Converter as separate module. It's main purpose is to generalize the hard-coded conversion. It would take QT3TS as Input and generate pure JSONiq Test Suite as output.

For implementing Test Converter we decided to create following classification of test-cases:

Emphasize that for Converter itself we need a plugin architecture and draw it. So that you can include and exclude some conversion.

Dependency name	Status
higherOrderFunctions	supported
moduleImport	supported
arbitraryPrecisionDecimal	supported
schemaValidation	not supported (XML specific)
schemaImport	not supported (XML specific)
advanced-uca-fallback	not supported
non_empty_sequence_collection	not supported yet
collection-stability	not supported yet
directory-as-collection-uri	not supported yet
non_unicode_codepoint_collation	not supported
staticTyping	not supported yet
simple-uca-fallback	not supported
olson-timezone	not supported yet
fn-format-integer-CLDR	not supported yet
xpath-1.0-compatibility	not supported (XML specific)
fn-load-xquery-module	not supported yet
fn-transform-XSLT	not supported yet
namespace-axis	not supported (XML specific)
infoset-dtd	not supported (XML specific)
serialization	not supported yet
fn-transform-XSLT30	not supported yet
remote.http	not supported
typedData	not supported
schema-location-hint	not supported (XML specific)
calendar	not supported yet
unicode-version	supported
unicode-normalization-form	supported
format-integer-sequence	not supported yet
xsd-version	supported
xml-version	supported
default-language	only "en" supported
language	only "en" supported
spec	only "XT30+" not supported
limits	not supported yet

Table 3.4: Supported Dependency List

3. XQUERY/XPATH 3.* TEST SUITE(QT3TS)

1. Fails, as expected and should not be converted to JSONiq. It will never be supported
2. Fails, as expected since it is not supported yet
3. Fails, but can be rescued with simple conversion. Any simple conversion like removing the "fn" prefix
4. Fails, but can be converted to JSONiq. Any complicated conversion like XML to JSON
5. Fails, because it is bug in Rumble
6. Succeeds

With this classification, we want to reuse most of Phase 2 Implementation Results presented in Table 3.3.

If we compare above described classification with classification in Table 3.3, we can notice that Fail corresponds to Item 5. Item 6 corresponds to Success. Managed corresponds to Item 3. Item 2 corresponds to Unsupported and Dependency. Skipped correspond to Item 1.

Performing iterations in Phase 1, we want to distribute all Crashes into some of Item 4 or 1. Of course, it is in our interest to identify as many test-cases as possible as Item 4 and perform conversion in Test Converter. Everything that we cannot convert, we will classify as Item 1.

Items 1 and 2 will be excluded from Test Converter output. However, we also need to take into account that over time, as Rumble implementation improves, tests from Item 1 will be distributed into 4 other categories. Therefore, we want to make highly modular and extensible architecture. We have decided to maintain a list of conversions that need to be performed. This list will be compiled by using all results obtained so far. The architecture will work as a "plugin". It will allow us to specify what conversions we will perform and what we will include or exclude from output. The important design decision remaining is the Data Format of the Test Converted output.

JSONiq and Test Converter Data Format

The JSONiq extension to XQuery allows processing XML and JSON natively and with a single language. This extension is based on the same data model as the core JSONiq and is based on the same logical concepts. Because of the complexity of the XQuery grammar, the JSONiq extension to XQuery has a less pleasant syntax than the JSONiq core. . When designing the Test Converter, we could have decided to use either XML or JSON as the underlying language. However, as our Test Driver was already implemented in the previous phase and was expecting XML as input and using the before

maybe cite something

mentioned Saxon for parsing it, we have decided to keep the same language for output of the Test Converter.

3.4.2 Architecture

The overview of scenario described in 3.4.1 can be seen in Figure 3.3

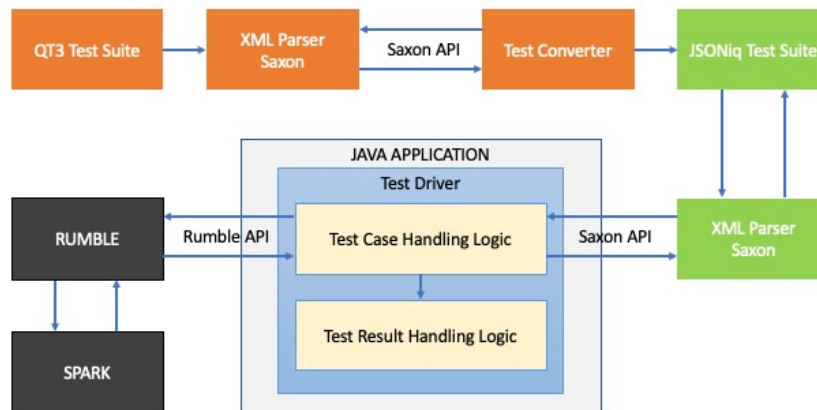


Figure 3.3: Phase 1 Architecture Overview

Chapter 4

Concluding Remarks

4.1 Overall Summary

sum up the results

4.2 Open Problems

further research here -
maybe web html page
of summary of the
results. Maybe auto
opening and closing
issues on Git

Bibliography

- [IFM⁺20a] Stefan Irimescu, Ghislain Fourny, Ingo Müller, Dan-Ovidiu Graur, Can Berker Çıkış, Renato Marroquin, Falko Noé, Ioana Stefan, Andrea Rinaldi, and Gustavo Alonso. Rumble supported error codes, 2017-2020.
- [IFM⁺20b] Stefan Irimescu, Ghislain Fourny, Ingo Müller, Dan-Ovidiu Graur, Can Berker Çıkış, Renato Marroquin, Falko Noé, Ioana Stefan, Andrea Rinaldi, and Gustavo Alonso. Rumble unsupported types, 2017-2020.
- [jso20] jsoniq.org. Jsoniq - the json query language, 2011-2020.
- [Kay16] Michael H. Kay. Qt3 test suite result summary, 2016.
- [Kay20] Michael H. Kay. Saxon: The xslt and xquery processor, 2020.
- [MFI⁺20] Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. Rumble: Data independence for large messy data sets, 2020.
- [Mih20] Stevan Mihajlovic. Rumble repository, 2020.
- [W3C11] W3C. Xquery/xpath/xslt 3.* test suite cvs repository, 2011.
- [W3C13] W3C. Xquery/xpath/xslt 3.* test suite, 1994-2013.
- [W3C20] W3C. Xquery/xpath/xslt 3.* test suite github repository, 2020.



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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):

A Test Suite for Rumble

Authored by (in block letters):

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

Name(s):

Mihajlovic

First name(s):

Stevan

With my signature I confirm that

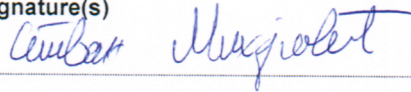
- I have committed none of the forms of plagiarism described in the '[Citation etiquette](#)' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

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

Place, date

Zürich, October 1, 2020

Signature(s)



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