ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@**ETH**_Zürich_

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

D INFK

Abstract

The increasing amount of data available to process in today's world led to the emergence of engines for large-scale data processing such as Spark. In order to make querying more efficient, Rumble is an engine that automatically runs queries on top of Spark, using JSONiq as declarative querying language instead of Spark API. Rumble is still in its beta version and requires a testing mechanism that would verify the behavior of its implementation. JSONiq inherits 95% of its features from XQuery, its XML counterpart. QT3TS is Test Suite with over 30000 test cases designed for testing XQuery. This work aims to create a Test Suite based on QT3TS to verify Rumble's implementation. As the first step, we have implemented a Test Driver capable of executing QT3TS on top of Rumble. The work was carried out gradually through several phases. In each phase, we were improving the implementation of both Test Driver and Rumble. Secondly, we have implemented a Test Converter for the XQuery to JSONiq conversion. As the final outcome of the work, we have managed to produce a JSONiq Test Suite that can be used to verify any JSONiq implementation, not just Rumble. We have made a significant impact on improving the implementation of the Rumble engine.

Acknowledgements

I am not writing this so that you can pity me. I am writing to share the story of how I learned to be grateful.

If you are thinking of quitting, do not do it. If you believe your life is miserable, it is not. Stop for a moment and look back at your life. Always look back at what you already have. Look at all the privileges you are entitled to, but you are taking them for granted. You came to this world as a tabula rasa, and the world owes you nothing by childbirth. Whatever you already have in life, you should consider a blessing. If life taught me anything: In a blink of an eye, you could lose everything.

In the end, it does not matter what you will achieve. What matters is what kind of life you are going to live. Live every day like it is your last and you will see how simple life is. And if you wake up one day, hating your life and the avatar that you have become, gather strength to reset and start all over again. To everyone that did terrible things to me, I can only say thank you. Thank you for making me play a game called life in veteran mode. You made me stronger. You made me what I am today. I died and I was reborn more than ten times. What is your excuse?

The list of people I am grateful to is endless, but I am limited to 2 pages. Whoever is left out should know that they still have a place in my heart:

- Nikola and Ivana - my brother and my mom, for being the only two people in the world for whom I would give my life

- Dusan Malinov - for implanting the idea of ETH in the first place

- Team 7 GameLab - for healing me after first-semester depression

- Daniel Keller - my boss at ABB who believed in my skills and capabilities that even when I did not believe in them

- Simon Weber - for pulling out more in me and not slowing me down

- Ex-wife - for teaching me what being manipulated actually means

- Ex-flatmate - for showing me how a toxic person can break a family

- Tommaso Ciussani - for saving me from abusive flatmate by sharing his room with me until I found a better place to live

- Costanza Importa - my teammate in CIL project for being human, having empathy and covering up for me when I lost my dad

- Regula Cinelli - for teaching me not to wait for time to pass in order to get something over with, but to picture yourself as a winner

- Svilen Stefanov - for being a true friend and my consigliere that I perhaps do not deserve

- Team Nignite - for heartwarming working environment and support

- Diana Ghinea - for seeing the value in me when I could not, for waking up desire and fire in me to be a better version of myself

- My dad Milutin - for telling the same stories ten and more times over and over again, teaching me that life is about memories

Contents

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 [25] is an engine that automatically runs queries on semistructured and unstructured documents on top of Spark, using the JSONiq language.

JSONiq [5] 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) [29] 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's implementation. The implementation of the Test Driver has been carried out gradually through iterations. We have analyzed the output of each iteration to measure the implementation success. Test Driver and its architecture have gradually evolved and improved through several phases of implementation. In parallel to Test Driver, we have helped improve Rumble implementation by filing bug reports. We have also designed strategies to automatically convert non-JSONiq test cases to JSONiq that led to implementing a Test Converter for the XQuery to JSONiq conversion.

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 underlying 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 the following three dimensions [11]:

- Volume - This term simply corresponds to the number of bytes that our data consists of. To have an idea of the scale, in Big Data, we are often looking at PBs of data. Scientific centers such as CERN produce tens of PBs 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 the underlying shape is a table. The main focus was on introducing normalization forms with the idea of avoiding data redundancy. Then the tables would 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

unstructured, nested, values are missing, etc. This trend led to Not Only SQL - NoSQL databases. The main focus in the NoSQL database is to perform opposite and 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 a 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 three main factors of this underlying medium drive are capacity, throughput and latency. From mid 1950's until today, we have witnessed a tremendous increase in all three factors. Capacity has increased by up to 200×10^9, throughput by 10×10^3 and latency by 8 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 than throughput, we need to read data from multiple medium drives simultaneously in parallel to obtain data fast enough. At the same time, to face the 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.

In summary, traditional Relational Database Management System - RDBMS, such as Oracle database or Microsoft SQL Server, have focused on being compliant with ACID (atomicity, consistency, isolation and durability) properties. Such RDBMSes with homogeneous tables are good when handling a small amount of data. However, if we need to scale the data massively, we need to turn to different technologies. These traditional RDBMSes that use file systems such as FAT32 or NTFS for physical storage are not sufficient [11].

On the other hand, NoSQL databases are compliant with the CAP (consistency, availability, partition tolerance) theorem. Examples of new NoSQL databases that have emerged are key-value stores (DynamoDB), document stores (MongoDb) and column-oriented stores (HBase). They often use Distributed File System (DFS) 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. These concepts lead to building high-performance and scale-able frameworks such as Hadoop that can query and process distributed massive quantities of data in parallel [11].

2.2 Hadoop

Apache Hadoop [33] 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 a cluster of commodity hardware. It consists of 3 components [11]:

- HDFS - Storage layer

- MapReduce - Processing layer

- YARN - Resource management layer

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

2.2.1 HDFS

The Hadoop Distributed File System - HDFS [27] is a physical storage layer of Hadoop inspired by Google File System - GFS [12] written in Java. It is one of the most reliable file systems 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 the scale of millions of PB files. This means that files are bigger than a single drive (medium). Therefore, in such a setting, the most suitable is block storage. Unlike a 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 three copies in case of a failure.

The architecture is master-slave. Master is NameNode and it is in charge of storing a namespace. The namespace is a hierarchy of files and directories. Since the blocks are 64 or 128 MB, the metadata is also small. And since we are storing a rather small amount of very large files, the whole namespace can fit in the RAM of the NameNode. In addition to the namespace, NameNode knows the file to blocks mapping together with the location of blocks and their 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 we have consistency and partition tolerance at the cost of availability from the CAP theorem. In case of a failure, there is a secondary NameNode that would start-up. Also, it enables high durability with three replicas and read/write performance. When reading, it usually transfers a lot of data - batch processing.

2.2.2 MapReduce

MapReduce [8] in its most broad definition is a programming model (style) answering the question of how we process the data. It consists of two crucial steps map and reduce alongside with shuffle as the intermediate step [11]:

- Map - Input data is mapped into an 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

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 page. This way, those 1000 pages can be done in parallel. The map task will perform Map $(K1, V1) -> List (K2, V2)$, where K1 is in the range from 1 to 1000 (for each page) and V1 is the text on each page. K2 will have values in the range of all possible words that occur in the document. V2 will always be 1. Such a mapper is very primitive. If reduce task is a commutative and associative function, then it is allowed to execute the same function in the map task to reduce the amount of shuffle that will happen afterward. As count is such a function, we can already perform sum per key in the map task. It means that K2 stays the same and V2 will be the actual count per page.

2. As not all possible words will appear on all pages, we will put together a 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 $(K2, List (V2)) -> List (K2, V3)$ - 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 a programming model can be used in any framework with any underlying 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 a 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 consists of multiple tasks, depending on how the data is split. And 1 task can be a map or reduce task. One or more tasks are then assigned to TaskTracker that needs to execute them. JobTracker is collocated with NameNode and TaskTracker, usually with the DataNode, in order to bring query to the data.

2.2.3 YARN

Yet Another Resource Negotiator - YARN [28] is a Resource Management layer in Hadoop Version 2. Comparing to Version 1, we might notice that JobTracker has many responsibilities. It is responsible for both scheduling and monitoring types of jobs. JobTracker is acting as the "Jack of all trades" and becoming a bottleneck in such a setting. Such bottleneck leads to scaleability issues and Hadoop could not handle more than 4000 nodes executing more than 40000 tasks (remember that job comprises a set of task).

The solution was introducing YARN that separates scheduling and monitoring responsibilities. The architecture is again master-slave where we have ResourceManager and NodeManager. It has a single ResourceManager per cluster in charge of only scheduling and performs: capacity guarantees, fairness, SLA, cluster utilization, assigns containers. It has a global overview of all cluster resources and provides leases for containers. One node in a cluster has one NodeManager and many containers. A container is an abstraction in which a task can be run and it comprises a set of resources such as RAM, CPU, storage, bandwidth that can be allocated to ApplicationMaster. ApplicationMaster has the responsibility to handle monitoring. In particular, it is in charge of: fault tolerance, monitoring, asking for resources, tracing job progress/status, heart-beating to the ResourceManager, ability to handle multiple jobs. We have many ApplicationMasters in a cluster, each job has one ApplicationMaster, but not every node has to have an ApplicationMaster. In essence, it can happen that a single node has multiple ApplicationMasters, each responsible for a different job completely unaware of the existence of other ApplicationMasters on the node. Finally, it should be noted that ApplicationMaster is a container. Described architecture solves the bottleneck issue allowing the cluster to scale up to 10000 nodes and 100000 tasks [11].

Full flow of duties overview:

- Clients submits a job.

- ResourceManager creates a job and returns ID.

- Client sends its requirements.

- ResourceManager tells a NodeManager to promote one of containers to ApplicationMaster.

- ResourceManager tells maximum capacity of containers.

- ApplicationMaster requests containers.

- ResourceManager assigns containers.

YARN offers a couple of types of schedulers, that based on application and its request in terms of resources, perform the allocation.

2.3   Spark

Apache Spark [35] [6] [18] is is an open-source engine for large-scale data processing. We see it as a generalization of MapReduce. From a straight pipeline of two tasks, map and reduce, it generalizes it to any Directed Acyclic Graph - DAG. DAGs are built around Resilient Distributed Datasets - RDDs [34] which are an abstraction for partitioned collection of values. On RDDs, we can perform creation, transformation and action. In Spark we need to make a clear separation of two plans, two graphs - lineage and DAG.

DAG is a physical plan of execution. A DAG is created when the user creates an RDD (by referencing a dataset in an external file system for example) and applies chains of lazy transformations on it. When action is called, it triggers the 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 them 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 [11].

Lineage graph tells us a logical plan. It tells us which RDD originates from which RDD. All the dependencies between the RDDs will be logged in the lineage graph rather than the actual data. This is called lazy evaluation, it only gets triggered when action is called. This lineage is used to recompute the RDD in case of failure.

Fault tolerance using lineage - Imagine that we start with an RDD on which we need to perform a couple of transformations and finally an action. Such RDD would first get partitioned so that multiple nodes can handle it. Imagine that some node fails, it means that only the partitions located on that node have to be recomputed. And the lineage graph is telling us exactly which set of transformations is needed to reconstruct the RDD.

DataFrame is a high-level abstraction of RDD's. It is a logical data model that enables users to view and manipulate data independently of physical storage. DataFrames store data in a collection of rows enabling the 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 underlying physical storage.

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

2.3.1 Apache Spark vs .Apache Hadoop MapReduce

For emphasizing the power of Spark, we have found a nice comparison with Hadoop MapReduce that can be separated into the following categories:

- Performance - Hadoop MapReduce stores the output on the disk after each map or reduce task. Spark keeps everything in memory. Spark performs better if all data is stored in RAM. If RAM is full, Spark uses disk, but overall it is better.

- Ease of use - Spark has compatible API for Python, Scala, Java. On the other hand, Hadoop MapReduce is written in Java, and it is hard to learn the syntax for programming.

- Cost - Spark needs a lot of RAM, so it is more expensive. All data needed for a job has to fit in RAM

- Data processing - Spark can do graph, ML, batch, and real-time processing, making it one platform for everything. Hadoop MapReduce is good for batch processing, but it does not support graph or real-time processing.

- Fault tolerance - Hadoop MapReduce relies on hard drives. In case of failure, it can continue wherever it left off and save time. It also has replication for fault tolerance. Spark uses RDDs for fault tolerance. They can refer to any dataset in external storage like HDFS. If RDD is lost, it is recomputed using transformations.

2.4 Data Format and Querying Language

2.4.1 JSON

JavaScript Object Notation - JSON [2] is a text-only, human-readable data format. It originates from JavaScript, but today it is a widely spread language-independent data format supported by many programming languages.

As we have seen, DataFrames in Spark and table data shape in general, which can be stored in CSV data format, is not suitable for heterogeneous data and de-normalization does not work. On the other hand, tree data shape and JSON as the data format is the perfect choice for nested heterogeneous data. It supports nesting by using two structured data types:

- Object - a collection of key-value pairs that act as an associative array (map) from string to any other type

- Arrays - ordered sequence of items of any type.

JSON also supports the four atomic data types that can be String, Number, Boolean and Null.

To better understand why JSON is good for nested data, we will provide an example. The code below shows that the value of the address is a nested JSON object consisting of two more nested JSON objects. Also, skills that have multiple values are presented using a JSON array.

```
{
        "firstName": "Stevan",
        "lastName": "Mihajlovic",
        "birthday": "1993-06-22",
        "nationality" : "Serbian",
        "address": {
                "city": {
                        "name": "Glattbrugg",
                        "code": "8152"
                },
                "country": {
                        "name": "Switzerland",
                        "code": "CH"
                }
        },
        "skills": [ "C#", "Java", "Python" ],
        "car": null,
        "isStudent": true
}
```

2.4.2 JSONiq

JSONiq [9], as mentioned in the introduction, is a declarative and functional querying language created exactly to analyze files written in JSON data format. It is designed to analyze tree-shaped data - nested and heterogeneous. It inherits 95% of its features from XQuery, its XML counterpart. It has a data model that can capture all aspects of JSON data format.

We say it is declarative because the user does not need to be aware of the underlying structure. It is a query language like SQL in the RDBMS, with the difference that it operates on JSON.

When it comes to the data model, everything is expressed as a Sequence of Items. Item itself can be any of the 6 data types that JSON supports. In addition, an Item can also be a Function. Then all Expressions that exist operate only on a Sequence of Items.

We say it is functional because Expression takes a Sequence of Items as the input and produces again a Sequence of Items as the output. It means that Expressions can be nested in any desired way.

The Expressions can be organized in the following categories:

- Arithmetic
- Logic
- Comparison
- Literal
- JSON construction
- JSON navigation
- Sequence construction
- Built-in function

- FLWOR expression
- Variable references
- Context item references
- Named function references
- Control flow expressions
- Type expression
- Static function calls
- Dynamic function calls

FLWOR expression is the most powerful. Using its own clauses, it is capable of everything Select From Where in SQL does - Selection, Projection, Grouping, Ordering, Join. It can also be nested any number of times in almost any order which SQL does not quite support [10]. To fully understand the power of FLWOR expression, we will provide an example. It is pretty declarative and intuitive for everyone who had a chance to write an SQL query before. We first select persons living in Glattbrugg. Then we are grouping them by nationality. Finally, we count persons in groups and select only those with more than 1000 to be displayed ordered. We can see freedom in writing this query - where can be applied multiple times, also let can be placed anywhere.

```
for $person in collection("personreg.json")
where $person.address.city.name eq "Glattbrugg"
group by $nationality := $person.nationality
let $count := count ($person)
where $count gt 1000
order by $count descending
return {
        nationality : $nationality,
        total : $count
}
```

Each clause produces a tuple stream in the FLWOR expression. It is a set of key-value pairs representing a binding from variable name to the corresponding Sequence of Items. The clauses can consume these tuple streams and produce tuple streams. So between themselves, clauses communicate via tuple streams. Since all Expressions operate on Sequence of Items, return clause that always has to be included in every FLWOR expression will actually consume tuple steam and produce Sequence of Items [10].

2.5 Rumble

Rumble is a query execution engine for large, heterogeneous, and nested collections of JSON objects built on top of Spark [25]. In this section, we will explain Rumble from the user perspective, also mappings from JSONiq to Spark via Rumble and General Architecture of Rumble.

2.5.1 User Perspective

The user can use Rumble via command line or using the Rumble API for Java. The architecture overview is quite simple and presented in Figure 2.1. A user only sees JSONiq query language and uses it to write the desired query. Rumble then takes this query and it has logic capable of mapping and passing the query down to Spark. Spark is then able to execute the query in the cluster. Spark usually reads from DFS, most typically HDFS we mentioned before. More in general, it can run on any file system or database. Typical input for a query is a JSON Lines document. JSON Lines document uses JSON data format and the only difference from a typical JSON document is that every line in the document is a single object. Such a document has a bit lower human-readability for nested data compared to JSON document, but it is quite commonly used in other fields such as Web Programming. [24]

Figure 2.1: Rumble Architecture Overview

2.5.2 Mapping

We said that Rumble has a logic that is capable of mapping the query to Spark primitives. We also said that in JSONiq, everything is a Sequence of Items. Therefore, Rumble uses interface Item in the code [15]. All six types mentioned in Section 2.4.1 implement this interface. After that, the Item is wrapped using the Spark JavaRDD generic class or DataFrame if the Sequence of Items is structured and the mapping is complete. Spark is now able to execute queries using objects of the wrapper class.

As also said, FLWOR Expressions are the most powerful ones and we can view them as a set of clauses. Between themselves, clauses operate by consuming tuple streams instead of operating on Sequence of Items. Sequence of Items is produced only in the end with the mandatory return clause. Therefore, in the code [15], Rumble uses class FlworTuple for wrapping to the Spark Dataset generic class that is used for DataFrames. For each clause, we have

a RuntimeTupleIterator and they all, with the exception of return, have a reference to FlworTuple. More details in Subsection 2.5.3.

2.5.3 General Architecture

So far, we were referring to Rumble as an engine. Essentially it is a compiler implemented in Java, and as such, it follows basic Compiler Design principles. In order not to break the declarative property of JSONiq query language, it requires proper separation of concerns. Irimescu in his thesis [14] proposed the layered architecture described in Figure 2.2. It consists of four main phases:

1. Lexer and Parser take JSONiq query as an input and produce Abstract Syntax Tree - AST as the output

2. Translator takes the AST as the input and produces a tree of expressions - Expression Tree as the output

3. Generator takes Expression Tree as input and converts it into a tree of runtime iterators - Runtime Iterator Tree

4. Runtime iterators represent the code that can be executed on a single node or on top of Spark

| JSONiq Query | Lexer &Parser | AST | Translator | Expression Tree | Generator | Runtime Iterator Tree |
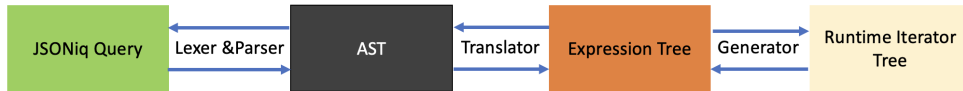
Figure 2.2: Rumble General Architecture

Lexer and Parser

The first steps in analyzing source code, in this case query written in JSONiq query language, are Lexical and Syntax analysis's performed by Lexer and Parser modules respectively. For rather simple languages, such as JSONiq, these two modules can be automatically generated from language grammar. Thus, Another Tool for Language Recognition - ANTLR v4 framework [26] is used. ANTLR needs a grammar (.g4) file with definitions of all language constructs as the input. For Rumble, JSONiq.g4 file was implemented and using it ANTLR auto-generated Parser and Lexer together with BaseVisitor (implements visitor pattern) Java classes. In the code, we can first use the Lexer class that takes JSONiq query stream as input and then pass it to Parser class which will generate AST and conclude the so-called "front-end" part of the compiler.

Translator

In general with compilers, AST cannot be used directly. As explained in [7], JSONiq is a functional language that is composed of expressions. Thus, higher-level abstractions - Expression Tree is needed. To achieve higher-level abstractions, the following classes had to be implemented. On top of the inheritance tree, we have abstract class Node from which Expression and Clause classes are derived. Clause class is then used for deriving all clauses of FLWOR Expression. For all other Expression categories mentioned in Section 2.4.2, classes were derived from the Expression class.

The second part of generating the Expression Tree required specific implementation of BaseVisitor class generated by ANTLR. BaseVisitor is a generic class and its specific implementation - TranslationVisitor class wraps around Node class.

The third part of generating the Expression Tree is the Static Context class containing a map between variable names and sequence types. Each Expression has its own Static Context.

Using all these classes, it is then possible to generate the Expression Tree as explained in [14]:

"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 time).

3. For any variable declaration it creates a new Static Context containing the new variable and sets the previously existing Static Context as a parent."

Generator

So-called "back-end" - the last part of the compiler includes code generation where the intermediate code gets transformed into assembly instruction and finally machine instructions. For this step in Rumble, we are performing conversion from Expression Tree to a tree of runtime iterators. As Rumble was written in Java, runtime iterators are in charge of executing operations that get converted to Java bytecode.

All RuntimeTupleIterator implement RuntimeTupleIteratorInterface while all other runtime iterators implement RuntimeIteratorInterface. Both interfaces are similar to java.util.Iterator interface with methods such as hasNext() and

next(). Using next(), runtime iterators can iterate over a Sequence of Items and return results one Item at a time. In addition, next() method triggers the computation of all children iterators by recursively calling next() method in them. Result of such implementation is "lazy evaluation", where results are computed only when demanded.

These two runtime interfaces operate on Dynamic Context containing a map between variable names and actual Sequences of Items. Static Context is in charge of static type checking performed at compile-time, while Dynamic Context is in charge of dynamic type checking performed at runtime.

As pointed out in [25] "The main novelty in the design of Rumble is the runtime iterators that can switch dynamically between different execution modes, and which encode the decision of which nesting level is distributed." In total, there are three different execution modes. In local execution mode, runtime iterators are executed on a single node locally in Java and they do not push computation to Spark. The other two, RDD-based execution (which uses Spark's RDD interface) and DataFrame-based execution (which uses the DataFrame interface), are executed on top of Spark. They both push computation to Spark when a dataset is large and there is a clear advantage over local execution mode. The modes that a runtime iterator supports are based on the JSONiq Expression category and presented in Table 2.1. The classification as we have seen it before in Section 2.4.2 is now translated into actual keywords sorted into five different categories. Each category supports different combination of execution modes. The columns L, R and D represent local, RDD-based and DataFrame-based execution modes, respectively, while + signifies that this mode is supported.

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

Table 2.1: Runtime iterator categorization for JSONiq expressions and clauses

Local-only iterators executed in the local execution mode come down to implementing the Expression's behavior in Java. On the other hand, RDD and DataFrame-based execution modes require a mapping to Spark primitives as explained in Section 2.5.2. There is an essential difference between these two modes that are running on top of Spark. The DataFrame-based mode is used in the case that internal structure is known statically. This mode is also preferred over RDD-based mode as it faster in execution. On the other hand, the RDD-based mode is used whenever the structure is unknown.

Rumble, in its initial version, was using RDD-based mode for FLWOR Expressions. However, all FLWOR Clauses Iterators, with the exception of return, operate with FlworTuple. From the query, it is possible to derive the static type of the variables in the tuples and therefore represent them as columns in DataFrame. Today, the RuntimeTupleIterator is using SQL queries instead of RDD transformations of Spark. We will not explain the mappings for each and every FLWOR clause in detail, but we will make a parallel to for clause and reuse an example from [25]. If the current variables are x, y, and z, and the new variable introduced by the for clause is i, then the for clause is mapped to the following:

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

Spark's EXPLODE functionality corresponds to flatMap() on RDDs, while UDF is a Spark SQL user-defined function that takes all existing variables as input and returns the resulting Sequence of Items as a List<Item>.

Chapter 3

Test Driver for Test Suite

3.1 Analysis

In this chapter, we will discuss design decisions that we have made during
Test Driver development. 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 try to view Rumble as a black-box whenever possible. Ideally, the single
point of communication with Rumble should only be via the Rumble Java
public API. In addition, as Rumble is also written in Java, the best option is
to implement Test Driver as Java Console Application. In the early Phases
of Implementation, it might not be possible to view Rumble as a black-box
completely. Thus, we will setup our Java Console Application project to have
two modules - Test Driver and Rumble module. Rumble module is the branch
in the repository created for the purpose of this work [21]. Including Rumble
as a separate module and making the Test Driver module dependent on it, we
allow the possibility to directly use Rumble and its classes in case that not
everything can 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 [30]. Since
April 1st 2019, the CVS tree has been discontinued and the repository has been
migrated to the W3C Public GitHub repository [32]. The tests are published
as a set of files - test sets containing in total more than 30000 test cases written
in XML format. W3C does not supply a Test Driver for executing the tests.
Instead, for each implementation, a Test Driver should be written [29]. As
these test sets are 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 the following possibilities:

- DOM (Document Object Model) - This parser loads the entire XML document in memory and uses interfaces to access the information. It can access a couple of item elements at the same time. It can be used for both reading and writing.

- SAX (Simple API for XML parsing) - This parser does not load XML document in memory. Instead, it allows us to register a handler with a SAX parser. When the parser goes through the file, it keeps invoking methods on the handler class for each item. It processes it in sequence one at a time. For each new item it reads, it forgets the state of previous items. Therefore, on each read, we need to take appropriate action in our application. It is read-only and it is also known as a push parser. There is no handler on the XML document side, only in our application.

- STAX (Streaming API for XML parsing) - This parser allows us to read and write multiple documents simultaneously. Unlike SAX that reads one item at a time, STAX can be explicitly asked to get a certain item from the XML document without loading it in memory. Therefore, we can look at it as a mixture of DOM and SAX. It is a pull parser and has a 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. It provides access to XSLT transformation, schema validation, and XPath processing services.

- Saxon [20] - 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 backward-compatibility mode.

  2. XPath processor. This supports XPath 2.0 and XPath 3.1. It can also be used in backward-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 four 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, the main arguments are:

1. Saxon itself is one of the implementations for which Test Driver was also implemented. Based on Results Report [19], it passes more than 99,9% of the QT3TS tests and it is considered a reference for XML.

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 underlaying 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:

```xml
<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 query that should be executed on Rumble. It can be XSLT, XPath or XQuery expression.

- Result - this is the expected result of the test tag. As it can be seen in the provided example, there are several types of assertions that we need to verify.

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. Next, we iterate over test cases in each of the test sets. For each test case, we are explicitly asking Saxon XML parser to get items under Test and Result tags. To use Saxon API, we need to know the structure. But, once the 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 an empty sequence

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

```xml
<test>math:cos(math:pi() div 2)</test>
<result>
    <assert>abs($result) lt 1e-15</assert>
</result>
```

- assert-eq - It requires us to run another query in the form of obtained result "eq" value under this 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 - Requires obtained result sequence size to be equal to this

- not - It requires to execute nested assertion with a negation

- assert-permutation - Requires result sequence to be permutation of this

- assert-xml - Requires result to be an XML document matching this one

- serialization-matches - Requires serialization of result to match this

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, the result obtained via Rumble API was compared with the expected result by applying the correct assertion check. In case assertion passed, a 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. An important note is that due to time limits, two assertions were not implemented. Thus, assert-xml and serialization-matches will always result in Crash. With this implementation, we were to be able to distinguish three possible scenarios:

1. Success - Test case succeed

2. Fail - Test case failed because of a bug in Rumble

3. Crash - Test case failed because it is not compatible with Rumble

The report is generated as a .csv file having test sets as rows and the 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

21

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 four major pain points:

- Unstable implementation of assertion which resulted in implementing a proper way of result binding in Rumble

- Too many crashing tests which resulted in implementing the converter

- Improving Test Driver implementation resulted in breaking previously implemented features. Therefore, regression tests were introduced

- Some tests were supposed to Crash with expected error codes and our granularity was not appropriate for distinguishing all test cases

Result Binding

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

```java
private boolean AssertEq(List<Item> resultAsList,
        XdmNode assertion) throws UnsupportedTypeException {
    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 UnsupportedTypeException{
    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 the first one. It does not handle sequences. Furthermore, handling sequences was only possible for AssertStringEqual in case that our result is a sequence of strings by performing string concatenation. All other assertions such as Assert, AssertEq, Assert-DeepEq are not possible to be implemented. Finally, if we remember Assert example from Section 3.2.1, we will notice that we had to use String.replace method replacing $result with actual result obtained from the Rumble API.

Thus, Rumble implementation itself was extended to support result binding. The only modification required in Test Driver was to instantiate a new RumbleConfiguration and also new Rumble instance for each test case that requires result binding. We then declare external variable and pass the obtained result to the newly created (nested) XQuery expression. Check the code below:

```java
private boolean Assert(List<Item> resultAsList,
        XdmNode assertion) throws UnsupportedTypeException {
    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 many new instantiates might cause the execution time to increase dramatically. However, run-time increased only by 15seconds from 2minutes - only 12.5%.

In Phase 1 Implementation, for AsserType, we had a switch case for every possible type that Rumble's Item class supports. Type information was obtained using methods of the Item class, which introduced a dependency on Rumble. Such code would be complex for future maintenance and extension with new supported types. Once the result binding was implemented, it allowed us to run the AssertType also as an "instance of" query. It also allowed to have a single point of the conversion performed in the beginning and applied for both the test case and the expected result. Within conversion, we would discover the unsupported type errors without the need of switch case to check whether Rumble's Item class supports the type or not. Furthermore, the previously implemented switch case had an 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 implementation using nested XQuery expression initialized idea and was a base plan for XQuery to JSONiq conversion logic separation. In Section 3.4.1 we will describe architecture with a separate module that takes XQuery as input, performs the conversion, and outputs JSONiq test suite. Such an 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 Rumble currently supports was taken from official Rumble documentation [17] and hard-coded 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, xs, 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 3.6 that was taken from [16].

Regression Tests

During Phase 1, we were performing iterations with the goal to improve Test Driver's implementation overall. The good metric while performing these iterations was the total number of test case Crashes. Our goal was to reduce those numbers as much as possible. This was mainly handled by making the following changes: bug fixes, software enhancements, configuration changes. Creating such 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) is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change. If not, that would be called a regression [3]. 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 made two comparisons 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 the opposite check and created list of "Tests that were not crashing before, but are now and not in list above".

Handling Error Codes and better granularity

In Section 3.2.1, we have described all the possible assertions for which we can verify whether the result of the executed query matches the expected result. However, some of the test cases are different. They are there to verify that certain queries cannot be parsed or executed because they are not compliant with the XML language. To better understand, we provide an example below:

```xml
<test-case name="Literals036">
   <description>
      Test for invalid decimal literal
   </description>
   <created by="Mike␣Rorke" on="2005-02-03"/>
   <test>65535032.001.01</test>
   <result>
      <error code="XPST0003"/>
   </result>
</test-case>
```

The issue is that a decimal number cannot have more than one decimal separation character (dot). Such a query should not parse, and there is a correct error code that should be reported instead - XPST0003 - "It is a static error if an expression is not a valid instance of the grammar defined in."

The issue in Test Driver implementation is that when such a query is passed down to Rumble, it will cause a RumbleException which breaks the execution of the code. Meaning that such queries need to be handled differently. Essentially, in case that executed query causes RumbleException, we need to check whether it has error codes as the expected result. If not, it will be considered a Crash. If it has an error code as the expected result, we need to check whether it matches the RumbleException's error code. Often, test cases have several error codes that they would accept under any-off assertion in the result tag. This is exactly the check that we are performing. In case that we have a single match with any of the possible error codes, we consider the test case to be a Success. They are there to verify that certain queries cannot be parsed or executed because they are not compliant with the XML language. Otherwise, it will go into the category Unsupported Error. The two tags for which we are not verifying assertion but checking error codes are:

- error - the code is under attribute code and starts with X

- assert-serialization-error - similar as error but starts with S

The complete classification diagram is shown in Figure 3.2.
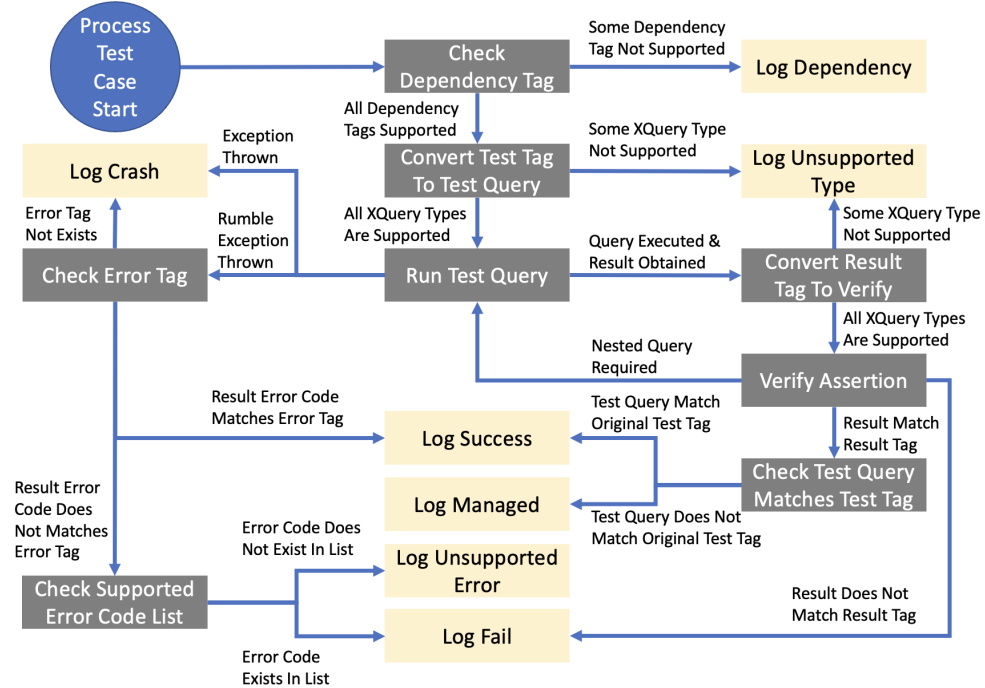
Figure 3.2: Phase 2 Classification Diagram

3.3.2 Architecture

The overview of scenario described in 3.3.1 can be seen in Figure 3.3
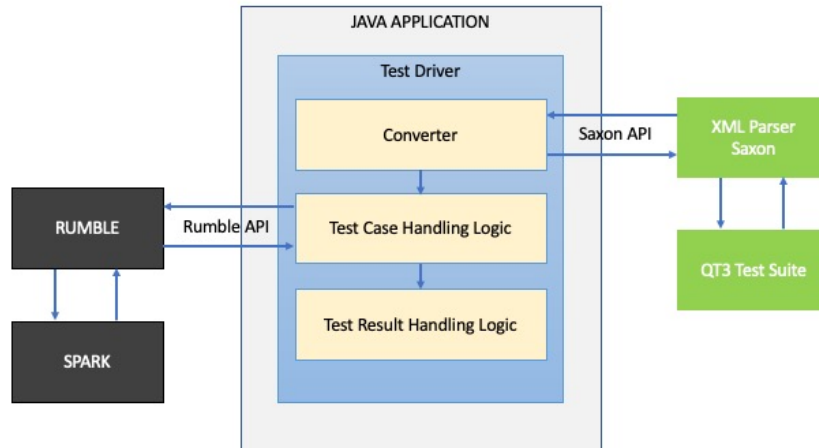
Figure 3.3: Phase 1 Architecture Overview

3.3.3 Results

As we have seen in previous sections, Crashes were not only capturing tests that are not JSONiq and needed conversion. They also include the tests that could not succeed because they contained type or error code that Rumble does not support. Such tests were now classified as Unsupported. Furthermore, some of them were introducing dependencies. For example, in the Dependency tag, it was possible to have a request for a particular version of XPath, XQuery or XSLT. While Rumble is backward compatible with all versions of XPath and XQuery, it does not support XSLT. Dependencies that were found in Test Suite were documented and classified according to Rumble documentation. From all dependencies, we have compiled into a list presented in Table 3.7. We have therefore created and divided test cases into six 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. Failed – Tests that are failing because there is a bug in Rumble API or Test Driver implementation.

4. Dependency – Tests that are failing because dependency is not supported

5. Unsupported – Tests that are failing because type or error code is not supported yet

6. Crash – Any other exception

After the introduction of the six 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
Dependency	1481	4.70
Crash	13171	41.80
Unsupported	7412	23.52

Table 3.2: Phase 2 Results Overview

Managed category was introduced as it was identified that with simple hard-coded conversion, we could obtain around 4200 passed tests increasing the total percentage of passed tests by roughly 12%. At first, it seems that Success and Managed should be grouped into a single category, but we decided to keep them separated. We want to keep track of Managed ones because in Phase 3 Implementation we are planning to generalize the hard-coded conversion and create a pure JSONiq Test Suite based on given XML ones.

As mentioned earlier, during Phase 1 we were performing iterations and going through all the Crashes, trying to eliminate them completely. 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 Rumble did not yet support the XQuery function or it had bugs, so Rumble implementation was also improved. We also identified both test cases that fail but can be converted to JSONiq and those that cannot be converted.

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 a separate module. Its main purpose is to generalize the hard-coded conversion. It would take QT3TS as the input and generate pure JSONiq Test Suite as the output.

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

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 a 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 a bug in Rumble

6. Succeeds

With this classification, we want to reuse most of Phase 2 Implementation Results findings and also classification presented in Table 3.2.

If we compare the above-described classification with classification in Table 3.2, we can notice that Item 5 corresponds to Fail, Item 6 to Success, Managed to Item 3. Item 2 to both Unsupported and Dependency.

Performing iterations as mentioned in Section 3.3.3, enabled us to identify and distribute Crashes into Item 4 or Item 1 category. It was in our interest to identify as many test cases as possible as Item 4 and perform the conversion in Test Converter. Everything that we could not convert due to any constraints, we classified as Item 1.

Items 1 will be excluded from the Test Converter output. Items 2, on the other hand, will be excluded from Test Driver input. However, we also need to take into account that over time, as Rumble implementation improves, tests from Item 2 will be distributed into four other categories. Therefore, we want to make highly modular and extensible architecture. 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 [5]. When designing the Test Converter, we could have decided to use either XML or JSON as the underlaying language. However, as our Test Driver was already implemented in the previous Phase and was expecting XML as input and using the before mentioned Saxon API for parsing it, we have decided to keep the same language for the output of the Test Converter.

3.4.2 Architecture

The overview of scenario described in 3.4.1 can be seen in Figure 3.4
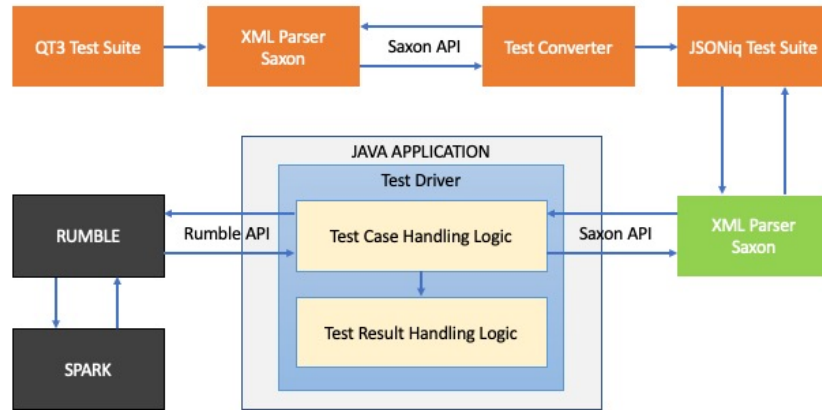
Figure 3.4: Phase 1 Architecture Overview

3.4.3 Results

After carefully analyzing the complete QT3TS, we have concluded that out of
424 test sets QT3TS consists of, a total of 143 can be classified as belonging to
Item 1 or Item 2 of classification documented in this Phase of Implementation.
More specifically, we have assigned 61 test sets to Item 1 presented in Table
3.4 and 84 test sets to Item 2 presented in Table 3.5.

In addition to test sets assigned to Item 2, we are adding the Unsupported
and Dependency. Furthermore, several bugs were fixed in Rumble between
Phase 2 and Phase 3 Implementation and results are presented in Table 3.3.

Scenario	Total test cases	% of all test cases
Item 1 (not JSONiq)	3675	11.65
Item 2 (not supported yet)	12751	40.41
Item 3 (simple conversion)	7007	22.20
Item 4 (can be converted)	4424	14.02
Item 5 (bug in Rumble)	999	3.17
Item 6 (succeeds)	2701	8.56

Table 3.3: Phase 3 Results Overview

fn/base-uri.xml	prod/AxisStep.ancestor-or-self.xml
fn/doc.xml	prod/AxisStep.following.xml
fn/document-uri.xml	prod/AxisStep.following-sibling.xml
fn/element-with-id.xml	prod/AxisStep.preceding.xml
fn/generate-id.xml	prod/AxisStep.preceding-sibling.xml
fn/has-children.xml	prod/AxisStep.static-typing.xml
fn/id.xml	prod/AxisStep.unabbr.xml
fn/idref.xml	prod/BoundarySpaceDecl.xml
fn/innermost.xml	prod/CompAttrConstructor.xml
fn/in-scope-prefixes.xml	prod/CompDocConstructor.xml
fn/json-to-xml.xml	prod/CompCommentConstructor.xml
fn/lang.xml	prod/CompElemConstructor.xml
fn/name.xml	prod/CompNamespaceConstructor.xml
fn/namespace-uri.xml	prod/CompPIConstructor.xml
fn/namespace-uri-for-prefix.xml	prod/CompTextConstructor.xml
fn/nilled.xml	prod/ConstructionDecl.xml
fn/node-name.xml	prod/ConstructionDecl.schema.xml
fn/outermost.xml	prod/Comment.xml
fn/parse-xml.xml	prod/CopyNamespacesDecl.xml
fn/parse-xml-fragment.xml	prod/DirAttributeList.xml
fn/path.xml	prod/DirectConstructor.xml
fn/resolve-QName.xml	prod/DirElemConstructor.xml
fn/root.xml	prod/DirElemContent.xml
fn/xml-to-json.xml	prod/DirElemContent.namespace.xml
xs/token.xml	prod/DirElemContent.whitespace.xml
op/except.xml	prod/NameTest.xml
op/intersect.xml	prod/NodeTest.xml
op/is-same-node.xml	prod/SchemaImport.xml
prod/AxisStep.xml	prod/StepExpr.xml
prod/AxisStep.abbr.xml	prod/ValidateExpr.xml
prod/AxisStep.ancestor.xml	

Table 3.4: Item 1 - Fails, as expected and should not be converted to JSONiq. It will never be supported

fn/compare.xml	map/size.xml
fn/analyze-string.xml	map/put.xml
fn/collation-key.xml	map/remove.xml
fn/contains-token.xml	map/for-each.xml
fn/data.xml	array/append.xml
fn/default-collation.xml	array/filter.xml
fn/default-language.xml	array/fold-left.xml
fn/environment-variable.xml	array/fold-right.xml
fn/escape-html-uri.xml	array/for-each.xml
fn/filter.xml	array/for-each-pair.xml
fn/fold-left.xml	array/get.xml
fn/fold-right.xml	array/head.xml
fn/for-each.xml	array/insert-before.xml
fn/for-each-pair.xml	array/join.xml
fn/format-integer.xml	array/put.xml
fn/format-number.xml	array/remove.xml
fn/function-lookup.xml	array/reverse.xml
fn/function-arity.xml	array/sort.xml
fn/function-name.xml	array/subarray.xml
fn/implicit-timezone.xml	array/tail.xml
fn/iri-to-uri.xml	xs/dateTimeStamp.xml
fn/load-xquery-module.xml	xs/error.xml
fn/local-name.xml	xs/normalizedString.xml
fn/local-name-from-QName.xml	xs/numeric.xml
fn/namespace-uri-from-QName.xml	op/bang.xml
fn/parse-ietf-date.xml	op/QName-equal.xml
fn/parse-json.xml	prod/Annotation.xml
fn/prefix-from-QName.xml	prod/BaseURIDecl.xml
fn/QName.xml	prod/ContextItemDecl.xml
fn/random-number-generator.xml	prod/ContextItemExpr.xml
fn/resolve-uri.xml	prod/DefaultCollationDecl.xml
fn/sort.xml	prod/DefaultNamespaceDecl.xml
fn/static-base-uri.xml	prod/EQName.xml
fn/unparsed-text.xml	prod/ExtensionExpr.xml
fn/unparsed-text-available.xml	prod/ModuleImport.xml
fn/unparsed-text-lines.xml	prod/NamedFunctionRef.xml
map/merge.xml	prod/NamespaceDecl.xml
prod/MapConstructor.xml	prod/OptionDecl.xml
map/contains.xml	prod/OptionDecl.serialization.xml
map/find.xml	prod/UnaryLookup.xml
map/get.xml	prod/VersionDecl.xml
map/entry.xml	prod/WindowClause.xml

Table 3.5: Item 2 - Fails, as expected since it is not supported yet

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 3.6: Rumble Supported Types and Error Codes

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 3.7: Rumble Supported Dependency List

Chapter 4

Test Converter for Test Suite

Figure 3.4 represents architecture that proposes implementing Test Converter as a separate module. The main idea driving us was having a clean architecture that would make the Test Driver easily maintainable and extensible. Thus, we will separate these two even more and implement them as two different Java console applications. Test Driver will only handle executing the test cases on Rumble. Test Converter will only handle converting the XQuery test suite to a JSONiq one, basically, it covers orange blocks from Figure 3.4.

However, in Section 3.4.1, we have explained the downsides of hard-coded conversion using Java String.replace method. Just having the clean architecture would not solve the hard-coding issues in Test Converter. Ultimately, we want to produce purely JSONiq Test Suite (similar to QT3TS for XQuery). Such Test Suite would be available for everyone to use and verify other JSONiq implementations such as Zorba [36], IBM WebSphere [13], Xidel [4].

4.1   Rumble Architecture

First, we need to think about achieving a clean XQuery to JSONiq conversion without hard-coding. Let us recall Rumble and its entire Section 2.5.3. We said that in order to finally execute the query, we are performing conversion from Expression Tree to a tree of runtime iterators. The Expression Tree itself is a higher-level abstraction of JSONiq functional language that is composed of expressions. The entire Node class hierarchy that was implemented in Rumble is used to represent the Expression Tree.

If we can create an Expression Tree from a query written in XQuery, it would mean that we have created JSONiq Expression Tree. Instead of creating the runtime iterator and executing the query, we can just serialize Expression Tree to JSONiq query directly. We will reuse Rumble and extend its architecture so that it supports parsing queries written in XQuery.

This means that even though Test Converter and Test Driver will be two separate Java console applications, they will both use Rumble. One may argue that the success of standalone JSONiq Test Suite will in that case depend on Rumble implementation. And this is true, it will. Nevertheless, the decision here is to use Rumble simply because using JSONiq Expression tree Rumble is the cleanest possible solution for conversion. Finally, it will also improve Rumble itself by supporting an additional query language - XQuery.

In Figure 4.1 we have represented that will enable us to achieve the target. In the following sub-sections, we will explain XQuery Lexer & Parser and Translator's implementation details together with the Serialization part.
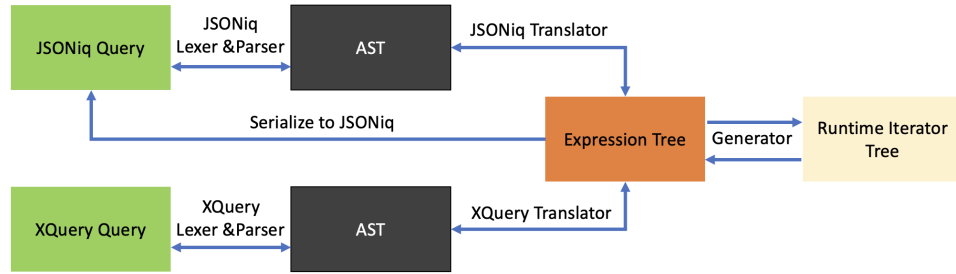

Figure 4.1: Rumble General Architecture with XQuery

4.2   Rumble Extension

Here we will go into details of the implementation of each and every modification that has been performed in Rumble in order for us to be able to reuse the already existing classes in the Node class hierarchy.

4.2.1   Lexer and Parser

If we recall Section 2.5.3, for simple languages, these two modules can be automatically generated from the grammar of the language. XQuery is 95% similar to JSONiq. Thus, we need to create a grammar file similar to JSONiq.g4.

The initial approach was to implement the XQuery.g4 file ourselves based on the already implemented JSONiq.g4 file in Rumble. However, XQuery is older than JSONiq and it was logical that such a file was already implemented. Indeed, we managed to find an ANTLR4 implementation of xqDoc for XQuery [1]. It is licensed under the same Apache License 2.0 as Rumble, and we can reuse it. The grammar is stable and it was not changed for more than 10 months, so we do not expect it to be updated. Also it is compliant with XQuery 3.1 W3C Recommendation [31] with couple of exceptions.

Names of certain labels in the grammar file were changed in order to match the code of TranslationVisitor class, which will be used as a baseline to implement XQueryTranslationVisitor. The structure remains the same except for:

1. module was not according to the W3C Recommendation [31] and allowed for multiple mainModules. See below the comparison between old and new module implementation.

   module : xqDocComment? versionDecl? xqDocComment? (libraryModule | (mainModule (SEMICOLON versionDecl? mainModule)*));

   module : xqDocComment? versionDecl? xqDocComment? (libraryModule | mainModule) ;

2. moduleDecl was changed to use uriLiteral instead of stringLiteral in order to match the implementation of TranslationVisitor class. This does not affect the structure as uriLiteral comes down to stringLiteral in the next level of nesting (uriLiteral: stringLiteral)

3. prolog was changed to use annotatedDecl in order to match the implementation of TranslationVisitor class. This does not affect the structure as annotatedDecl offers the same options that were originally in prolog:

   annotatedDecl: varDecl | functionDecl | contextItemDecl | optionDecl;

4. functionDecl and varDecl were not according to the W3C Recommendation [31]. It allowed for (annotations | ncName) while the ncName is under annotations but three levels below in nesting

5. varDecl was not according to the W3C Recommendation [31]. varValue and varDefaultValue were defined as expr instead of exprSingle and it allowed them to be surrounded by { }. See below the comparison between old and new varDecl implementation.

   varDecl: KW_DECLARE annotations KW_VARIABLE DOLLAR varName typeDeclaration? ((COLON_EQ varValue) | (KW_EXTERNAL (COLON_EQ varDefaultValue)?) | (LBRACE varValue RBRACE) | (KW_EXTERNAL(LBRACE varDefaultValue RBRACE)?)) ;

   varDecl: KW_DECLARE annotations KW_VARIABLE DOLLAR varName typeDeclaration? ((COLON_EQ varValue) | (KW_EXTERNAL (COLON_EQ varDefaultValue)?)) ;

   varValue: expr ; $->$ varValue: exprSingle ;

   varDefaultValue: expr ; $->$ varDefaultValue: exprSingle ;

6. squareArrayConstructor was changed to use expr instead of exprSingle (COMMA exprSingle)* in order to match the implementation of TranslationVisitor class. This does not affect the structure as it is equivalent in the next level of nesting (expr: exprSingle (COMMA exprSingle)* ;)

7. arrowExpr was changed to use complexArrow instead of arrowFunction-Specifier argumentList in order to match the implementation of TranslationVisitor class. This does not affect structure as it is equivalent in next level of nesting (complexArrow: arrowFunctionSpecifier argumentList;)

We have also implemented another script for handling the XQuery.g4, and using it, ANTLR auto-generated XQueryParser and XQueryLexer together with its corresponding XQueryParserBaseVisitor base class.

4.2.2 Translator

The second part requires us to implement XQueryTranslationVisitor that extends the generated XQueryParserBaseVisitor class and wraps around the Node class. With proper implementation, we would be able to achieve converting XQuery into the JSONiq Expression Tree. The implementation is mainly based on the already implemented JSONiq TranslationVisitor with modifications that we will document here.

Of course, not everything can be converted to JSONiq and it should not be converted. Below we will list the conversions that we have left out as they will never be supported:

- schemaImport

- copyNamespacesDecl

- constructionDecl

- boundarySpaceDecl

- optionDecl

- nodeComp

- unionExpr

- intersectExceptExpr

- parenthesizedExpr within the arrowFunctionSpecifier of arrowExpr

- URIQualifiedName

- validateExpr, extensionExpr within valueExpr of unaryExpr

- nodeConstructor within primaryExpr

- kindTest, typedMapTest, typedArrayTest within itemType

- axisStep within stepExpr

- multiple stepExpr within relativePathExpr

- single or double dash preceding relativePathExpr within pathExpr

In addition to that, there are conversions that we are yet not able to perform as the implementation is missing in Rumble. As Rumble implementation improves, so will the implementation of XQueryTranslationVisitor as well. For now, the following conversions are classified as out of the scope of the thesis and will be supported in the future once Rumble is upgraded:

- defaultNamespaceDecl

- decimalFormatDecl

- baseURIDecl

- contextItemDecl

- existUpdateExpr within exprSingle

- parenthesizedExpr and STAR object lookup within keySpecifier of lookup

- orderedExpr and unorderedExpr within primaryExpr

- windowClause within initialClause of flworExpr

- functionTest within itemType

For the versions that we support, since the grammar file is based on the XQuery 3.1 W3C Recommendation [31], we have decided to support XQuery versions 1.0, 3.0 and 3.1 as version 3.1 is backward compatible.

For annotations, currently, we only support single public annotations without prefix, since from QT3TS in the prod/ModuleImport.xml, tests modules-pub-priv-29 to 36, we can see that "It is an error if a variable's annotations contains any combination of two annotations". In addition, according to the W3C Recommendation [31] it is said that "If no prefix is present, the name is in the "http://www.w3.org/2012/xquery namespace". This results in the following implementation of annotation handling:

1. Throw an error if there is more than one annotation in the annotations list

2. Otherwise, extract the EQName

3. Throw an error if prefix is not equal to http://www.w3.org/2012/xquery

4. Otherwise, throw an error if local name is not public

5. Otherwise, let the query with annotations be converted

Namespaces fn and map that were removed in Section 3.3.1, we are now binding. Namespace xs is part of grammar. For namespaces map and array, no binding is needed as they are counted for Item 2 as mentioned in Section 4.3

Handling the Literals is slightly different. If we compare JSONiq with XQuery, we will see that true, false and null literals do not exist in XQuery. Instead, they will be added as function calls.

The itemType of XQuery covers more possibilities compared to the JSONiq:

itemType: kindTest | (KW_ITEM LPAREN RPAREN) | functionTest | mapTest | arrayTest | atomicOrUnionType | parenthesizedItemTest ;

itemType : Kitem | atomicType; | Kobject | Karray | Kjson;

In JSONiq TranslationVisitor implementation, it was enough to call getItemTypeByName method and pass the parsed context. In XQuery implementation, this is only possible for the atomicOrUnionType. For all others, we have to handle it differently. First of all, kindTest will throw an error as it corresponds to the seven complex (non-atomic) types that should not be converted (document, element etc.). FunctionTest is not yet supported in Rumble and it is out of the scope of the thesis. The mapTest and arrayTest correspond to AtomicItemType.Object and AtomicItemType.Array item type of JSON if they are untyped, otherwise we throw an error. Item is the AtomicItemType.item while parenthesizedItemTest recursively calls the same method.

When it comes to arrayConstructor, there is a slight problem. JSONiq and XQuery do not have the same data model - there is a data model mismatch. XQuery allows us to have a Sequence of Items as array members or object values. JSONiq does not support that and requires single items instead. The only way to store a Sequence of Items in the JSONiq data model is using arrays. In order to overcome the dependency mismatch, we introduce mapping between two data models and we map nested Sequence of Items of XQuery to JSONiq array. In the case of squareArrayConstructor of XQuery, we recursively iterate over the exprSingle in expression and perform visitExprSingle. Each expression returned by visitExprSingle, is then wrapped into an array constructor. Finally, all these array constructors are wrapped into Comma Expression that is then wrapped in the main array constructor. In the case of curlyArrayConstructor, we first obtain the content by calling visitExpr. Then we instantiate a new Context Item Expression and wrap it into an array constructor. We combine these two by instantiating a new Simple Map Expression and wrap it in the main array constructor.

4.2.3 Serialize to JSONiq

Instead of running the query and returning the iterator over the resulting Sequence of Items, we need to perform serialization. Serialization is basically an inverted process of parsing. We have declared a new method serializeToJSONiq in the abstract class Node. All other classes used to create the JSONiq Expression Tree are derived from Node, and in those classes, we need to implement the method. We are now peeking into the XQuery.g4 grammar file in order to recreate the String representation of each and every Expression. In each implementation, we use the appropriate keywords and recursively call serializeToJSONiq for each nested Expression.

In Section 2.5.3, we have presented the 4 phases that Rumble goes through as a compiler. If we further analyze these phases through the code of Java Rumble API, we can identify the following steps that are required in order to execute the query:

1. Instantiate Lexer from the input stream - this is the complete query

2. Instantiate Parser using the Lexer from Step 1

3. Instantiate Static Context (map between variable names and sequence types) using the URI file - this is the input file on which the query is executed

4. Instantiate Translation Visitor using Static Context from Step 3

5. Instantiate MainModule by calling visit method of Translation Visitor and passing the main module extracted by the Parser as the argument

6. In the runQuery method of Java Rumble API, the first five Steps are performed by calling parseMainModuleFromQuery method that returns Main Module from Step 5

7. Instantiate Dynamic Context (mapping between variable names and actual sequences of items) using returned Main Module from Step 6

8. Instantiate RunTimeIterator using returned Main Module from Step 6

9. Return the Sequence of Items result instantiated using the Iterator from Step 8 and Dynamic Context from Step 7

To perform only serialization without obtaining the query results, we have extended Java Rumble API with an additional method serializeToJSONiq. This method performs the same six first Steps described above. After that, we simply call the implementation of serializeToJSONiq of abstract Node class via Main Module from Step 6. We pass String Buffer as the argument, which gets populated and then the Java Rumble API serializeToJSONiq method returns it as a String. Such implementation allowed Rumble to provide conversion out of the box via JSONiq Expression Tree.

4.3 Architecture

As we said before in Section 3.4.1 we will distinguish six cases, but for this implementation we will need only first two:

1. Fails, as expected and should not be converted to JSONiq. It will never be supported. - Documented in Table 3.4

2. Fails, as expected since it is not supported yet - Documented in Table 3.5

This separation is important to understand. As we want to produce a standalone JSONiq Test Suite similar to QT3TS for XQuery, all Item 1 should not be included as they are not JSONiq. On the other hand, all Item 2 should be included because they can be used to verify other JSONiq implementations.

The final architecture of the Test Converter can be seen in Figure 4.2. There is a slight difference compared to Test Driver architecture. Test Converter does not need to execute the JSONiq query. Therefore as presented in Figure 4.1, we will simply call serializeToJSONiq instead of creating runtime iterators that would be executed on top of Spark, i.e. we do not need Spark.
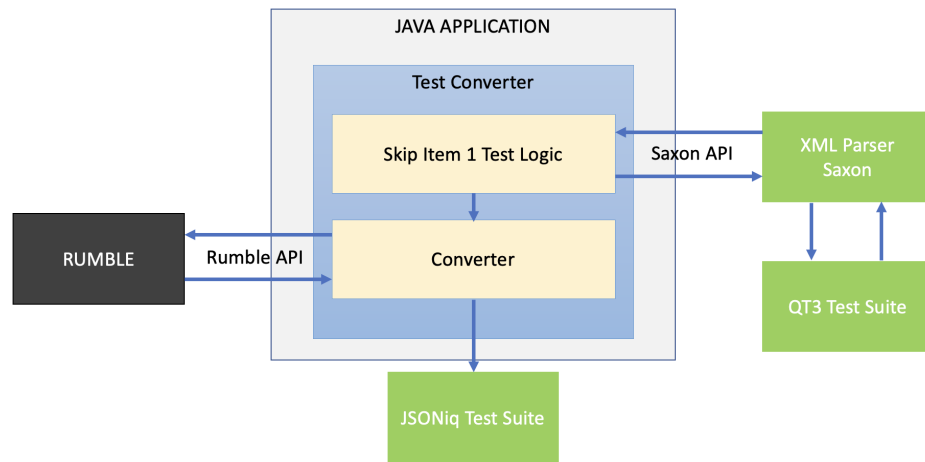

Figure 4.2: Test Converter Final Architecture

Final Test Driver architecture can be seen in Figure 4.3. We will maintain a list of Item 2 that will not be executed in Test Driver for Rumble. Test Driver will now take JSONiq Test Suite produced by Test Converter and execute it.
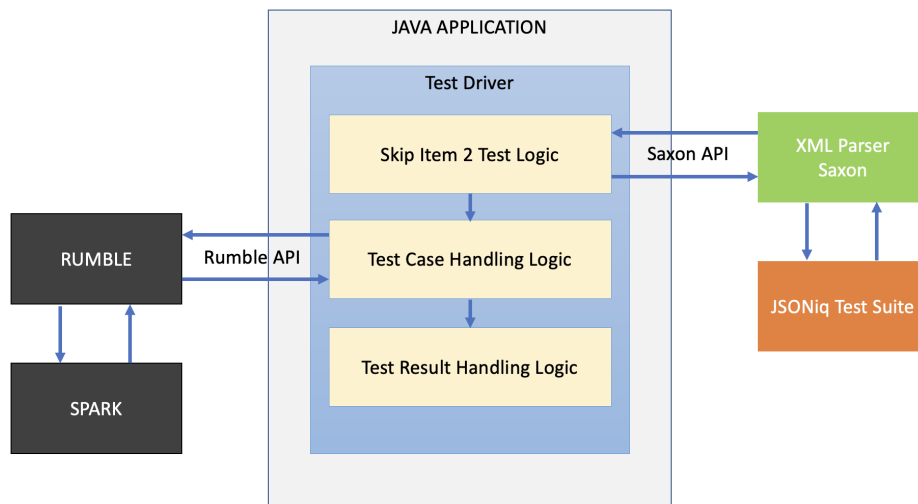

Figure 4.3: Test Converter Final Architecture

4.4 Implementation

The Converter part is using method binding provided by Saxon. First, we define the ExtensionFunction and the name under which it will be available in XQuery expression (convert). Secondly, we specify what the result of this function is (the result of the Java Convert method). Finally, we assign it a namespace (bf). We now declare XQuery expression that has a function that recursively visits all nested XML tags. In case that they are test or result tags, it will pass the underlaying tag value to bf:convert and replace the tag value with the returned result. We declare external variable and pass the entire test set to the XQuery expression. Check the code below:

```java
Processor testDriverProcessor = new Processor(false);

ExtensionFunction converter = new ExtensionFunction() {
    @Override
    public QName getName() {
        return new QName(bindingNameSpace, "convert");
    }

    @Override
    public SequenceType[] getArgumentTypes() {
        return new SequenceType[]{
            SequenceType.makeSequenceType(
                ItemType.STRING, OccurrenceIndicator.ONE
        )};
    }

    @Override
    public XdmValue call(XdmValue[] arguments) throws SaxonApiException {
        String arg = arguments[0].itemAt(0).getStringValue();
        String result = Convert(arg);
        return new XdmAtomicValue(result);
    }

    @Override
    public SequenceType getResultType() {
        return SequenceType.makeSequenceType(
            ItemType.STRING, OccurrenceIndicator.ONE
        );
    }
};

testDriverProcessor.registerExtensionFunction(converter);
XQueryCompiler xqc = testDriverProcessor.newXQueryCompiler();
xqc.declareNamespace("bf", bindingNameSpace);
XQueryExecutable xqe = xqc.compile(
    "declare function local:transform($nodes as node()*) as node()*{\n" +
        "for $n in $nodes return\n" +
            "typeswitch ($n)\n" +
                "case element (test) " +
                    "return <test>{bf:convert($n/string())}</test>\n" +
                "case element (result) " +
                    "return <result>{bf:convert($n/string())}</result>\n" +
                "case element () " +
                    "return element { fn:node-name($n) } " +
                                "{$n/@*, local:transform($n/node())} \n" +
                "default " +
                    "return $n\n" +
    "};" +
    "declare variable $test-set external;\n" +
    "let $y := $test-set//test-set\n" +
    "return local:transform($y)");
XQueryEvaluator xQueryEvaluator = xqe.load();
xQueryEvaluator.setExternalVariable(new QName("test-set"), testSetDocNode);
xQueryEvaluator.iterator();
```

The Java Convert method simply uses Rumble API to call serializeToJSONiq method we have previously created, making the conversion complete. There is a small remark to take into consideration here. The test sets that have error codes will cause a RumbleException. Due to time constraints, we could not make a better classification and check for which error codes we need to perform XQuery to JSONiq conversion of test tag. Thus, those test sets will not be converted and return the same result instead. Check the code below:

```java
private String Convert(String testString){
    try {
        return rumbleInstance.serializeToJSONiq(testString);
    } catch (RumbleException re) {
        return testString;
    } catch (Exception e){
        return testString;
    }
}
```

At the moment, Test Converter is dependent on Rumble as it was used to perform clean XQuery to JSONiq transformation. However, once Rumble is mature implementation supporting everything we have documented as missing, we will have a once and for all generated JSONiq Test Suite as our work output.

On the other hand, Test Driver will not change much. We have a similar architecture as we had with the hard-coded conversion before but in a much cleaner approach. Test Driver can be used directly with either JSONiq or XQuery Test Suite depending on configuration, which we will explain in Section 5.1. It will be helpful in improving the Rumble implementation by fixing and enable opening/closing issues and tracking the development progress of Rumble. Fixing the bugs and adding features will also consequentially lead to Item 2 list dropping down to zero.

Chapter 5

Conclusion and Future Work

As mentioned in Chapter 1, 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. However, during our work, we have managed to go beyond this scope and achieve even more, which we will present in Section 5.1.

5.1 Result Summary

Apart from the implementation of the Test Driver, the results can be divided into a total of four major areas:

1. Implementation of Test Driver for Rumble

2. Improvement of Rumble implementation

3. XQuery Parser extension of Rumble

4. Standalone JSONiq Test Suite

5.1.1 Implementation of Test Driver for Rumble

As we said, the published QT3TS Repository [32] can be used to test any XML (XQuery, XPath, XSLT) implementation. Rumble as an engine uses JSONiq language that inherits 95% of its features from XQuery. For each implementation, Test Driver has to be written in order to be able to use the QT3TS. And we have achieved that. We have a fully operational Test Driver that can parse the QT3TS and execute it on top of Rumble. Depending on the configuration, it can be used in three modes:

1. The preferred way of testing Rumble implementation uses the original QT3TS and performs hard-coded conversion to JSONiq within Test Driver. Once Rumble implementation is mature enough, it will consequently lead to a mature JSONiq Test Suite. Then using QT3TS will become obsolete, while using JSONiq Test Suite will be preferred.

2. Verify implementation of XQuery Parser of Rumble by using the original QT3TS without performing the hard-coded conversion to JSONiq within Test Driver.

3. Future way of testing Rumble implementation is using the JSONiq Test Suite generated from original QT3TS by Test Converter without performing the hard-coded conversion to JSONiq within Test Driver.

5.1.2 Improvement of Rumble Implementation

In this subsection, we will discuss the impact and the usage that Test Driver had on Rumble implementation. Let us take another look at Table 3.2. This is one of the versions in which Test Driver itself was not very stable as it still had bugs in the implementation. The fully stable version after which we had code freeze on Test Driver was implemented on 12th January 2021. After this version, we have performed a manual inspection of failed and crashed test cases. We have opened over 50 issues on Rumble Repository [23]. Here, we will omit the test cases that were skipped and aggregate categories in order to present simple classification as we had in Table 3.1. In Table 5.1 we can see how did Rumble engine improve over the period of 2 months by implementing bugfixes of the 50 above-mentioned issues. Column # presents total number of test cases per category while column % presents the percentage the category takes in subset of all non skipped test cases.

Result from 12.01.21			Result from 12.03.21		
Scenario	#	%	Scenario	#	%
Success	8837	58.4	Success	9764	64.5
Fail	1351	8.9	Fail	999	7.6
Crash	4938	32.7	Crash	4372	28.9

Table 5.1: Rumble Implementation Improvement

The Crashes that are still visible in Table 5.1 can be improved by further classification of test sets and test cases into Item 1 or Item 2 that should be skipped as explained in Section 4.3. It can also be improved by extending Test Driver to support assert-xml and assert-serialization-matches.

5.1.3 XQuery Parser Extension of Rumble

So far, Rumble was able to use only JSONiq as querying language. In order to convert the QT3TS - XQuery Test Suite to JSONiq, we decided to reuse the JSONiq Expression Tree already existing in Rumble. We first implemented XQuery Parser and XQueryTranslationVisitor that enabled us to obtain the JSONiq Expression Tree from query written in XQuery. We then implemented serialization that takes the JSONiq Expression Tree and outputs query written in JSONiq. The byproduct of such an exercise resulted in extending the Rumble such that it can now operate using XQuery language as well.

5.1.4 Standalone JSONiq Test Suite

One of the most significant achievements of our work is producing purely JSONiq Test Suite similar to QT3TS for XQuery. This Test Suite uses a .xml file format similar to QT3TS and it can be published and used by anyone to verify their JSONiq implementations. Anyone could write their own Test Driver and use our JSONiq Test Suite in a similar fashion as we used the QT3TS one.

5.2 Future Work

Throughout this work, we have discussed many ideas and developed many prototypes. The prototypes are fully operational. However, they can be extended or improved. The open problems that remained unresolved are:

- Test Driver and Test Converter - Extending Test Driver and Test Converter to support the last two missing assertions: serialization-matches and assert-xml.

- Test Driver - Implement a separate class for outputting the results. Right now, all the outputs are in the form of log files in .txt or .csv file format. The step forward would be to implement a class that would form an HTML web-page similar to the one QT3TS has.

- Test Driver - extend Test Driver such that it can automatically detect bugs based on test cases that are not succeeding and automatically open/close issues on Rumble Repository [15].

- Test Converter - Improve serialization to JSONiq such that it has better file formatting with new lines and brackets.

- Test Converter - Some test cases are written in a way that they do not parse or cause other errors. Some of them should not be converted, and Test Converter can be extended not to convert certain test cases based on their expected error code.

- Rumble - Improve XQuery Parser such that it passes more test cases using the original QT3TS.

- Rumble - Enable Rumble to automatically detect the underlaying query language and use the appropriate JSONiq or XQuery parser in order to execute the query.

We will not present the code nor the instructions on how to use it. We will also not add it to the appendix. Everything is well documented on the git repository [22] created for the purpose of implementing this work.

Bibliography

[1] An Antlr4 implementation of xqDoc for XQuery. `https://github.com/xqdoc/xqdoc`. [accessed 21-Feb-2021].

[2] Introducing JSON. `https://www.json.org/json-en.html`. [accessed 12-Dec-2020].

[3] Regression testing. `https://en.wikipedia.org/wiki/Regression_testing`. [accessed 06-Dec-2020].

[4] Xidel. `https://github.com/benibela/xidel`. [accessed 21-Feb-2021].

[5] JSONiq - The JSON Query language. `https://www.jsoniq.org`, 2011-2020. [accessed 06-Dec-2020].

[6] B. Chambers and M. Zaharia. *Spark: The Definitive Guide Big Data Processing Made Simple*. O'Reilly Media, Inc., 1st edition, 2018.

[7] C. B. Cikis. Machine Learning with JSONiq. Master's thesis, ETHZ, 2020.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, CA, 2004.

[9] D. Florescu and G. Fourny. JSONiq: The History of a Query Language. *IEEE Internet Computing*, 17(5):86–90, 2013.

[10] G. Fourny. *JSONiq The SQL of NoSQL*. 2013.

[11] G. Fourny. ETHZ Big Data Lecture - 263-3010-00L, 2018.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, Oct. 2003.

[13] IBM Corp. IBM WebSphere DataPower Gateways Release Notes. https://www.ibm.com/support/knowledgecenter/en/SS9H2Y_7.5.0/com.ibm.dp.doc/releasenotes.html, 2017. [accessed 21-Feb-2021].

[14] S. Irimescu. JSONiq on Spark. Master's thesis, ETHZ, 2018.

[15] S. Irimescu, G. Fourny, I. Müller, D.-O. Graur, S. Mihajlovic, M. Arduini, C. B. Çıkış, R. Marroquin, F. Noé, I. Stefan, A. Rinaldi, and G. Alonso. Rumble GitHub Repository. https://github.com/RumbleDB/rumble, 2017-2021. [accessed 15-Feb-2021].

[16] S. Irimescu, G. Fourny, I. Müller, D.-O. Graur, C. B. Çıkış, R. Marroquin, F. Noé, I. Stefan, A. Rinaldi, and G. Alonso. Rumble Supported Error Codes. https://rumble.readthedocs.io/en/latest/Error%20codes/, 2017-2020. [accessed 26-Dec-2020].

[17] S. Irimescu, G. Fourny, I. Müller, D.-O. Graur, C. B. Çıkış, R. Marroquin, F. Noé, I. Stefan, A. Rinaldi, and G. Alonso. Rumble Unsupported Types. https://rumble.readthedocs.io/en/latest/JSONiq/#unsupportedunimplemented-features-beta-release, 2017-2020. [accessed 26-Dec-2020].

[18] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark: Lightning-Fast Big Data Analytics*. O'Reilly Media, Inc., 1st edition, 2015.

[19] M. H. Kay. QT3TS Saxon Result Summary. https://dev.w3.org/2011/QT3-test-suite/ReportingResults31/report.html, 2016. [accessed 06-Dec-2020].

[20] M. H. Kay. SAXON: The XSLT and XQuery Processor. http://saxon.sourceforge.net, 2020. [accessed 06-Dec-2020].

[21] S. Mihajlovic. Rumble Repository. https://gitlab.inf.ethz.ch/gfourny/rumble/-/tree/research-project-stevan-mihajlovic, 2020. [accessed 24-Dec-2020].

[22] S. Mihajlovic. Rumble Test Suite Repository. https://gitlab.inf.ethz.ch/gfourny/rumble-test-suite, 2020. [accessed 12-Feb-2021].

[23] S. Mihajlovic. Issues submitted by mstevan on Rumble GitHub Repository. https://github.com/RumbleDB/rumble/issues?q=is%3Aissue+author%3Amstevan+, 2021. [accessed 27-Mar-2021].

[24] I. Müller and G. Fourny. IRIS-HEP Topical Meeting (4 May 2020) - Rumble: JSONiq (query language) on Spark . https://www.youtube.

`com/watch?v=dbUQZbtfNfk&feature=emb_logo`, 2021. [accessed 15-Feb-2021].

[25] I. Müller, G. Fourny, S. Irimescu, C. B. Cikis, and G. Alonso. Rumble: Data Independence for Large Messy Data Sets. 2020.

[26] T. J. Parr and R. W. Quong. ANTLR: A Predicated LL(k) Parser Generator. *Softw. Pract. Exper.*, 25(7):789–810, July 1995.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[28] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013. Association for Computing Machinery.

[29] W3C. XQuery/XPath/XSLT 3.* Test Suite (QT3TS). `https://dev.w3.org/2011/QT3-test-suite/`, 1994-2013. [accessed 06-Dec-2020].

[30] W3C. QT3TS CVS Repository. `https://dev.w3.org/cvsweb/2011/QT3-test-suite/`, 2011. [accessed 06-Dec-2020].

[31] W3C. XQuery 3.1: An XML Query Language W3C Recommendation. `https://www.w3.org/TR/xquery-31/`, 2017. [accessed 21-Feb-2021].

[32] W3C. QT3TS GitHub Repository. `https://github.com/w3c/qt3tests`, 2020. [accessed 06-Dec-2020].

[33] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 4th edition, 2015.

[34] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, Apr. 2012. USENIX Association.

[35] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM*, 59(11):56–65, Oct. 2016.

[36] Zorba. Zorba NoSQL engine. `http://www.zorba.io/`, 2013. [accessed 21-Feb-2021].

ETH

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.

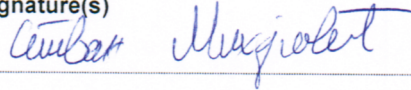| **Name(s):**	**First name(s):**
Mihajlovic	Stevan

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.