ETH zürich

ETH Library

Rumble DB Updates

Master Thesis

Author(s):
Loughlin, David

Publication date:
2023

Permanent link:
https://doi.org/10.3929/ethz-b-000634905

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Master's Thesis Nr. 445

Systems Group, Department of Computer Science, ETH Zurich

Rumble DB Updates

by

David Loughlin

Supervised by

Prof. Dr. Gustavo Alonso, Dr. Ghislain Fourny

August 2023

D INFK

Contents

1 Introduction **1**

2 Background **3**

2.1 Big Data . . . 3
2.2 Data Independence . . . 4
2.3 XQuery . . . 4
2.3.1 XML . . . 4
2.3.2 XQuery Data Model . . . 5
2.3.3 XQuery Update Facility 1.0 . . . 5
2.4 JSONiq . . . 8
2.4.1 JSON . . . 8
2.4.2 JSONiq Data Model . . . 8
2.4.3 JSONiq Processing Model . . . 9
2.4.4 JSONiq Updates . . . 10
2.5 RumbleDB . . . 16
2.5.1 Execution Modes . . . 16
2.5.2 From Query to Result . . . 17
2.5.3 AST Generation . . . 17
2.5.4 Expression Tree Generation . . . 17
2.5.5 Iterator Tree Generation . . . 18
2.5.6 Sequence Generation . . . 19
2.6 Persisting Updates . . . 21
2.6.1 Transactions . . . 21
2.6.2 Delta Lakes (Databricks) . . . 22

3 Implementation of Updates **24**

3.1 Grammar Extension . . . 24
3.1.1 Transform Syntax . . . 24
3.1.2 Function Declaration Syntax . . . 25
3.1.3 Update Locator Syntax . . . 25
3.1.4 Delete Syntax . . . 26
3.1.5 Replace Syntax . . . 26
3.1.6 Rename Syntax . . . 27
3.1.7 Insert Syntax . . . 27
3.1.8 Append Syntax . . . 27
3.2 Updating Expressions . . . 28
3.2.1 Expression Classification . . . 28
3.2.2 Expression Classes & Visitors . . . 29
3.2.3 Transform Expression Class . . . 30
3.2.4 Inline Function Expression Class . . . 30
3.2.5 Update Locator Processing . . . 30
3.2.6 Delete Expression Class . . . 31
3.2.7 Replace Expression Class . . . 31
3.2.8 Rename Expression Class . . . 31
3.2.9 Insert Expression Class . . . 32
3.2.10 Append Expression Class . . . 32
3.3 Updating Iterators . . . 32

3.3.1 Extensions to Runtime Iterators 32
3.3.2 Updating Iterator Classes 33
3.4 Sequences & Pending Update Lists 34
3.4.1 Update Primitives In RumbleDB 34
3.4.2 Pending Update Lists in RumbleDB 37
3.4.3 Mutability Scoping 44
3.4.4 Pending Update List Materialisation 45
3.4.5 Sequences in an Updating World 48
3.5 Delta Table Integration 49
3.5.1 Delta Files 49
3.5.2 Item Identifiers 50
3.5.3 Updating Delta Tables 52

4 Experimentation **57**
4.1 Structure 57
4.1.1 Datasets 57
4.1.2 Systems & Queries 58
4.1.3 Benchmarking Methodology 59
4.2 Performance Evaluation 59
4.2.1 Performance Metric 60
4.2.2 TPC-C Evaluation 60
4.2.3 GitHub Archive Evaluation 61
4.2.4 RumbleDB Evaluation 62

5 Conclusion **66**
5.1 Future Work 66

A Appendix **68**
A.1 Expression Classification Enum 68
A.2 Expression Classification Visitor Default Action 69
A.3 Standard Update Primitive Interface 70
A.4 Local Pending Update List Tree Map Comparator 71
A.5 Github Archive Schema 71
A.6 JSONiq Github Archive Delta Table Creation 72
A.7 Query 1A: JSONiq 72
A.8 Query 1A: SparkSQL 73
A.9 Query 1B: JSONiq 73
A.10 Query 1B: SparkSQL 73
A.11 Query 1C: JSONiq 74
A.12 Query 1C: SparkSQL 74
A.13 Query 2A: JSONiq 74
A.14 Query 2A: SparkSQL 75
A.15 Query 2B: JSONiq 75
A.16 Query 2B: SparkSQL 75
A.17 Query 2C: JSONiq 75
A.18 Query 2C: SparkSQL 76
A.19 Additional Performance Graphs 76

Abstract

In this thesis, we extend RumbleDB with the ability to process and persist updates for heterogeneous, and nested collections of JSON documents. RumbleDB is a query execution engine for large, messy, and heterogeneous data that implements the JSONiq query language and is built on top of Apache Spark. Semi-structured datasets are increasingly widespread and so is their need to be manipulated and updated. JSONiq provides a specification for the syntax and semantics for the expressions that enable updating JSON, but this specification is not yet implemented using the data independence offered by RumbleDB and so its potency is not realised. The key challenge in the design, implementation, and persistence of JSONiq updates is maintaining the correctness of the update to the state in spite of the arbitrary nestedness of both the JSON documents and the JSONiq queries. We demonstrate the extensions to the grammar, expression tree, iterator tree, and local execution model of RumbleDB that enable the creation and correctness of Pending Update Lists and six varieties of updating expressions as integrated into RumbleDB's arsenal. We also outline the incorporation of Databricks' Delta Lake API into RumbleDB allowing for the persistence of updates to a Big Data ready format, and we describe the best practices and logical models to use in order to extend our implementation. Our query experimentation and performance analysis compares our local execution of persisting updates to SparkSQL. These experiments show RumbleDB starts with a mean execution time four times worse than SparkSQL then worsens linearly as more data is processed. To rectify the inferior performance we detail optimisations that would drastically improve performance to be in line with SparkSQL. Nevertheless, our analysis showcases the viability of JSONiq updates and our implementation to be used as an intuitive and extensible model for further development. Our results illustrate that with data independence, updating arbitrarily nested and heterogeneous data can be simple, correct, and extendable.

1 Introduction

Data collection and processing have always been important aspects of our society, enabling numerous technological advancements and even the understanding of our very own genome (Clarke et al., 2012). Today they are more vital than ever before, but as we have advanced so has the data we need to analyse. In 2022, Amazon Glacier boasts storing exabytes of data for single customers alone (ama, 1991), while the entirety of Shakespeare's work has been collated by Motl and Schulte (2015) into 8.8 megabytes of relational tables in MariaDB (mar, 2019). In the face of such vast amounts of data, we must recall that even as little as 8.8 megabytes can have the ability to impact the entire world. Therefore, it is crucial that we do not discriminate between data, but rather embrace the Volume, Velocity, and Variety of data (Laney et al., 2001) that form the foundations of the Big Data paradigm.

Shakespeare's work was originally stored in a textual format making the use of relational tables feel strange as their rigid, tabular structure and homogeneity do not immediately lend themselves to encapsulating the plays, the acts within the plays, the speeches within the acts, and the rest of the creative structure that comprise Shakespeare's work. However, thanks to the NoSQL movement we need not limit our data to such unfitting formats. One popular format of NoSQL is JSON (Crockford, 2006). Intuitive, concise, and simple, JSON is used by numerous systems and information-sharing frameworks (being recommended as a core principle of the REST API by Rodriguez (2008))) to naturally represent all forms of nested and heterogeneous data. The ubiquity of JSON is thanks to its straightforward data model of documents made of arrays or sets of key-value pairs of atomic values. However, a standardised format is not enough to utilise data and as SQL did for the relational model (Codd, 1970), many query languages have sought to make the processing of JSON easy. SQL++ (Ong et al., 2014) and jaql (Beyer et al., 2011) are attempts to outline a query language capable of intuitively querying JSON data. Nevertheless, SQL++ falls short as it insists on being fully backwards compatible with SQL meaning it has to compromise, thereby making querying certain aspects of JSON (mainly arrays) less intuitive as they require additional clauses to transform them into a context SQL can understand. Moreover, in being as descriptive as possible jaql loses its sense of being a declarative query language and instead, each query resembles an enigmatic configuration file making the language feel more like an API. Rectifying these deficiencies is JSONiq (Robie et al., 2011), a declarative and functional query language inspired by an extending XQuery (Boag et al., 2002) to JSON data, but in actuality, JSONiq is capable of querying most semi-structured, tree-shaped data formats. JSONiq boasts an intuitive navigation syntax making traversing heterogeneous data easy, and when combined with FLWOR expressions JSONiq removes the complexity associated with expressing queries on deeply nested data.

Analysing data is only one facet of data management and has been the focus of much research into Big Data. However, static, immutable data has little use in a dynamic world. The On-Line Transaction Processing (OLTP) paradigm is defined by the need for systems to handle swathes of concurrent transactions, each of which may be updating the state of the data and so introducing a dynamic element into data accesses. Many OLTP systems rely on relational databases including PostgreSQL (Group) and mySQL (mys), thus they are restricted to tabular homogeneity. Nonetheless, there is no inherent reason that the NoSQL movement cannot handle OLTP requirements, and systems like MongoDB (Chodorow and Dirolf, 2010) do exactly that. The aspect of OLTP we will focus on in this paper is the ability to update records and with the advent of JSONiq updates (Robie et al., 2011) and JUpdates (Brahmia et al., 2022), it is clear that the need to update data extends into the realm of semi-structured data. Where JSONiq updates and JUpdate differ is in the shoulders upon which they stand. JSONiq updates aim to be naturally nested, like JSON itself, and so inherit from the similarly natively nested query language XQuery. Whereas, JUpdate aims to be more SQL-like, thereby removing some intuition behind the shape of JSON data. Therefore, JSONiq is capable of intuitively representing both the analysis and updating of semi-structured JSON-like data, where "updating" refers to the modification, deletion, insertion, and renaming of arbitrarily nested fields without whole-document overwrites.

JSONiq is only a specification for a query language and so does not facilitate semi-structured data processing alone. RumbleDB (Müller et al., 2020) is a query execution engine for JSONiq built on top of Spark (Armbrust et al., 2015) enabling the parallel processing of very large, messy distributed datasets. RumbleDB provides the connection between JSONiq and Big Data. The focal element of RumbleDB is the data independence it offers to decouple the logical processing model of JSONiq from the physical implementation of the execution provided by a local volcano-style model and the parallelism offered by Spark. However, RumbleDB does not yet offer data independence over JSONiq updates meaning a crucial aspect of data processing is lacking from the system. Thus, in this thesis, we extend RumbleDB by implementing the logical and physical components that will bring JSONiq updates into RumbleDB's data independence. Here we will outline the requirements and practices that enable the updating of Big Data's semi-structured data; in so doing we will showcase the viability and extensibility of the data independence that RumbleDB offers. To highlight the possibilities for JSONiq updates in RumbleDB, we will focus on implementing the core framework and models that facilitate the local execution of these updates while remaining easy to extend for future parallel execution. This means extending the suite of JSONiq expressions to include updating expressions, creating the data model that will represent the updates these expressions intend to enact, and then enacting these updates with complete correctness.

Once JSONiq updates have been integrated into RumbleDB, there still would not be any mechanism for keeping these updates. Does an update that is not persisted ever really update at all? We believe that the answer is no and so it is vital that RumbleDB provide a way to maintain and persist these updates. One such technology, built for transactional processing in an ACID-compliant manner while sustaining a connection to the cloud and the Big Data ecosystem, is Databricks' Delta Lake (Armbrust et al., 2020). MongoDB is another system capable of facilitating transactional processing and even natively does so for tree-shaped data like JSON. However, unlike MongoDB, Delta Lake and RumbleDB are united by their use of Spark and aim to widen the reach of data management in Big Data. Hence, this thesis' secondary aim of extending RumbleDB to persisting JSONiq updates is implemented by supporting Delta Lakes. In order to perform this extension we utilise RumbleDB's data independence to broaden the physical execution of JSONiq updates to be applicable to data derived from Delta Lakes, without making any changes to the logical interface of JSONiq.

Subsequently, RumbleDB will be able to update and persist the updates of arbitrarily nested heterogeneous data, with its consistency and correctness verified against the JSONiq updates specification (Robie et al., 2011), the XQuery specification (Consortium et al., 2011), and a suite of test and use cases in a CI/CD pipeline.

The remainder of this thesis comprises an investigation into updating the semi-structured data of Big Data. This investigation is structured in three overarching parts. Firstly, in Chapter 2 we present an overview of Big Data and data independence that will lead us to the technologies developed to meet their challenges, namely, the creation of JSONiq, RumbleDB, and Delta Lakes. Following this, in Chapter 3 we outline and discuss the extensions to RumbleDB's grammar, expressions, iterators, and processing that facilitate the integration of updates into RumbleDB. We conclude this section with our incorporation of Delta Lakes to persist updates processed in RumbleDB. We then present an analysis of experiments comparing RumbleDB and SparkSQL's processing of various updating queries in Chapter 4. Finally, concluding remarks and a roadmap for potential future work are offered in Chapter 5.

2 Background

2.1 Big Data

The adjective “Big” in Big Data is misleading as it does not simply refer to the vast amount of data produced in the modern day that requires storage and processing, but rather the “Big” refers to how great every aspect of data has become. With this outlook, it is far easier to understand the foundational categorisation of Big Data that (Laney et al., 2001) identify with the 3 V’s of Big Data. Each V highlights the dimensions of Big Data: great Volume, great Velocity, and great Variety. Volume describes the most intuitive attribute of Big Data – the size. Big Data scales far beyond even petabytes of storage, (Sagiroglu and Sinanc, 2013), and will only continue to grow. Velocity refers to both the speed at which data is produced and the need to be processed under the Big Data paradigm. Finally, Variety outlines the number of shapes in which Big Data appears, generally described as structured, semi-structured, and unstructured. Structured data is rigid, homogeneous, and analysable, the most prominent example of which is any data that fits under the tabular relational model. Unstructured data on the other hand is the exact opposite, it is random, hard to describe, and in a raw form like text, audio, or video. However, semi-structured data meets them in the middle by being denormalised, heterogeneous, and nested while maintaining a level of describability – tree-shaped formats, like JSON, and graphs, like RDFs, are the most notable examples. Each categorisation of Variety has its own subcategories, making the problem of being able to process the great variety of Big Data easy for some data and extremely hard for others, thereby adding another layer of complexity. Although there are other V’s that can be considered for Big Data, like Veracity in (Cappa et al., 2021), the foundational V’s described above will provide an ample understanding of the Big Data paradigm.

Each facet of Big Data poses a different problem for modern data management and processing that must be tackled by innovation. As a result of these problems, many different cloud-based computing architectures have been adopted to facilitate Big Data management, one such notable paradigm being the data lake. Data lakes are huge collections of datasets spanning a variety of storage systems and formats, with little useful metadata and a high degree of variation over time; consequently, their structures are often heterogeneous. Data lakes are routinely supported by well-established systems, namely Hadoop systems White, and fortuitously their implementation is analogous to data warehouses, leaving room to adapt data warehouse processing ideas, like metadata management and data population, to meet the needs of data lakes outlined by Cuzzocrea (2021). Thus it has been possible to quickly progress using established research from the old architectures. Data lakes then prove to be the answer to all of Big Data’s problems. The use of object stores, like S3 (Ama, 2002), and distributed file systems, like HDFS (Borthakur et al., 2008), allow data lakes to handle the volume of Big Data. Moreover, (Sawadogo and Darmont, 2021) outlines data ingestion services, like Flink (apa, a) and Kafka (apa, b), along with ETL (Extract Transform Load) methodologies that match Big Data input velocities, while processing techniques, like MapReduce (Dean and Ghemawat, 2008) and Spark (Zaharia et al., 2016), are used to keep up with the output velocity demands of Big Data in data lakes. Lastly, data lakes even tackle Big Data’s variety through extensive support for SQL and NoSQL storage solutions: relational DBMSs, like MySQL (mys), handle structured data; document stores, like MongoDB (Chodorow and Dirolf, 2010), and graph databases, like Neo4j (neo), facilitate semi-structured data; and aforementioned object stores to store unstructured data.

Big Data processing, however, is not limited to MapReduce and Spark as they themselves are limited to either key-value pairs or tabular formats. Moreover, neither provide a query language natively designed for semi-structured data, rather they rely on APIs or extensions to the natively relational query language SQL. Thus, to better encompass the variety of processing demanded by Big Data, many query languages have been developed to be native to non-tabular data shapes. Some such languages are JSONiq (Robie et al., 2011) for the tree-shaped data of JSON (Crockford, 2006), and Cypher (Francis et al., 2018) for the graphs of Neo4j. Only with the integration of native query languages can the power and complexity of semi-structured and unstructured data be found.

2.2 Data Independence

Data Independence, introduced for relational models in (Codd, 1970), is described as the independence of application programs and terminal activities from growth in data types and changes in data representations but can be summarised as the separation of logical and physical models in data systems. This understanding of data independence can be expanded to the technologies of data lakes described in section 2.1, in that the solutions they provide must be independent of one another to allow for better cohesion. Such a concept can be depicted by layering the technologies, with the divides between layers representing their independence from one another. Nevertheless, the core of data independence lies in the difference between the physical and logical data structures; this thesis looks at the technology interfacing between the user and the logical data structure – the query language. With data independence, a query language like SQL can process queries on the logical, relational model abstracted away from the underlying storage model, be it the N-ary storage model, the decomposition storage model (Copeland and Khoshafian, 1985), or the PAX model (Ailamaki et al., 2001). Thus, query processing engines need only to support SQL to ensure that the same query will return the same result regardless of the storage model used, thereby giving both query writers and database architects the freedom to work independently (Tsatalos et al., 1996).

Despite the importance of data independence in traditional data management systems, Big Data has struggled to take advantage of data independence (Markl, 2014), and so technologies like Spark-SQL of (Armbrust et al., 2015) have sought to stretch the independence capabilities of established query languages – SQL – into the heterogeneous data models of Big Data. This approach to the integration of data independence is taken one step further by establishing data independence in RumbleDB (Müller et al., 2020), a query execution engine implementing JSONiq from (Robie et al., 2011) which is a query language native to the tree-shaped data described in section 2.1. Utilising data independence to create more free, efficient workflows for users and developers that directly handle data in its natural form could enable rapid innovation and shielded workflows throughout the Big Data space; this thesis works to grant more of this independence by extending RumbleDB to handle updating JSON objects.

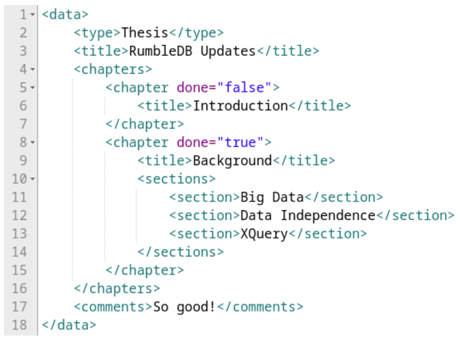
2.3 XQuery

XQuery, introduced in (Boag et al., 2002), is the precursor to JSONiq, the query language this thesis focuses on, and was designed to query and allow the processing of XML data in a concise and understandable manner. Much like its successor, XQuery is derived from another language, called Quilt (Chamberlin et al., 2000), and so has taken inspiration from non-XML query languages like SQL.

2.3.1 XML

XML, or eXtensible Markup Language, is a markup language with similar syntax and structure to HTML (Raggett et al., 1999). However, the primary goal of XML is to easily describe data for storage and communication. These data then comprise data objects dubbed XML documents, as outlined in (Bray et al., 1998).

The logical structure of XML is simple in that the main components of an XML document are elements, attributes, and text. These components are further specified by the XML information set (Infoset), described by (Consortium et al., 2004), in which XML documents are *Document Information Items*, elements are *Element Information Items*, attributes are *Attribute Information Items*, and text is



```
<data>
    <type>Thesis</type>
    <title>RumbleDB Updates</title>
    <chapters>
        <chapter done="false">
            <title>Introduction</title>
        </chapter>
        <chapter done="true">
            <title>Background</title>
            <sections>
                <section>Big Data</section>
                <section>Data Independence</section>
                <section>XQuery</section>
            </sections>
        </chapter>
    </chapters>
    <comments>So good!</comments>
</data>
```

Figure 2.1: Example of an XML document with nesting

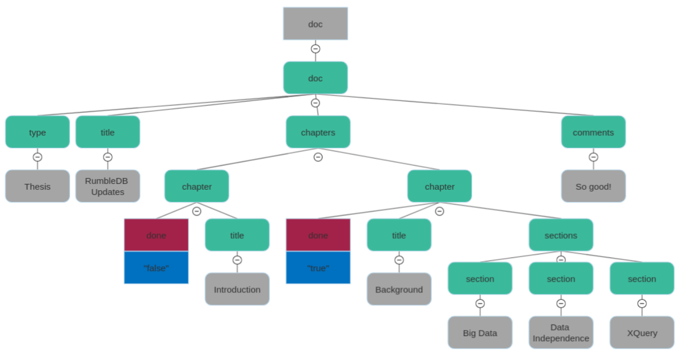


Figure 2.2: Tree structure representing the XML in figure 2.1

made up of many *Character Information Items*. Each information item has a set of properties, but for the purposes of this thesis, these properties only need to be described roughly. Elements have an opening tag and a closing tag with everything between the tags being tagged data – mainly text and nested elements. Attributes, on the other hand, are within the opening tags of elements and are the key-value pairs of an element.

Together, the information items of XML can represent a variety of data formats, while enabling arbitrary nestedness through elements – an example being 2.1. Thus, much like the JSON to come, XML is excellent at portraying tree structures in data as seen in figure 2.2.

2.3.2 XQuery Data Model

The XQuery data model shares concepts with the XML Infoset, especially when considering how to visualise the logical data structure, but it extends the type information of the Infoset and supports the notion of ordered, heterogeneous sequences (Consortium et al., 2017). In XQuery, every instance of the data model (an XDM instance) is a sequence, and sequences are simply ordered collections of zero or more items that cannot contain other sequences. The items within a sequence are either nodes, functions, or atomic values. Nodes are akin to the information items of the XML Infoset discussed in subsection 2.3.1, whereas functions are items that can be called. With XQuery being a functional language, all basic components of functions are present in that functions have names, parameters, type signatures, and a set of instructions that map parameters to some instance of the result type. Finally, atomic values are values that have an atomic type derived from the types of XML schema outlined in (Biron et al., 2004), but for the purposes of this thesis, these atomic types can be assumed to be limited to the simple atomic types found in a programming language like C.

Given the complex nestedness and heterogeneity of XML, the processing of millions of XML documents demands the robust sequence-oriented data model of XQuery; it is from this foundation that the processing and updating of the simpler data format, JSON, can be approached.

2.3.3 XQuery Update Facility 1.0

As specified by (Consortium et al., 2011), XQuery has an update facility that provides expressions that can be used to make persistent changes to instances of the XQuery data model.

The basic building block of XQuery is the expression, and from here various expressions can be composed with one another to form more complex queries. Nevertheless, every expression has zero or more input XDM instances and returns a single XDM instance as a result. Under this processing model, no expression can modify the state of existing XDM instances beyond creating new instances. However, the update facility introduces the notion of an updating expression, which can modify the state of existing instances. To accomplish this, five new expressions are added – delete, insert, replace, rename, and transform – and all expressions are categorised as either simple, basic-updating, updating, or vacuous where:

- **Simple expressions** are any expression that is not an updating or basic-updating expression.
- **Basic-updating expressions** are any delete, insert, replace, or rename expressions, or a call to an updating function.
- **Updating expressions** are any basic-updating expression, or any expression directly containing an updating expression (recursively defined), except transform expressions
- **Vacuous expressions**: any expression explicitly defined to be vacuous by XQuery Update Facility 1.0 – nevertheless they all either return an empty XDM instance or raise an error.

Introducing updating expressions means the processing model is extended so that the result of an expression consists of both an XDM instance and a *Pending Update List (PUL)*, either or both of which may be empty, meaning that both cannot be non-empty. PULs are unordered collections of update primitives, which represent the change of state intended to be applied to a specified XDM instance. Two update routines act on PULs: `upd:mergeUpdates` merges the update primitives of two PULs, and `upd:applyUpdates` enacts the changes described by the update primitives of a PUL. Moreover, `upd:applyUpdates` is implicitly invoked when the outermost expressions in a query returns a PUL. Finally, the update primitives of a PUL are under strict snapshot semantics, meaning they are resolved on the state of items before any update of the PUL is applied. `upd:mergeUpdates` can only act on PULs of the same snapshot, while `upd:applyUpdates` terminates a snapshot. In the XQuery update facility, each entire query is defined as one snapshot.

Both XQuery and JSONiq have specifications for updates, and both share the vast majority of semantics as JSONiq is derived from XQuery and will be discussed later in subsection 2.4.4. Thus, to avoid repetition, only the extensions not overridden by JSONiq updates, namely transform expressions and updating functions, will be described in this subsection.

Syntax 2.1: Transform Expressions

The general form of a transform expression is:

```
TransformExpr ::= "copy" "$" VarName ":=" ExprSingle
                    ( "," "$" VarName ":=" ExprSingle )*
                          "modify" ExprSingle "return" ExprSingle  (2.1)
```

Transform expressions create modifiable copies of existing XDM instances with new, unique identities. These expressions then return an XDM instance that may contain items that existed prior to the transform expression or were created and modified by it. Most notably, however, transform expressions do not modify the value of existing instances and so the expression is simple.

The keywords, `"copy"`, `"modify"`, and `"return"` denote the three clauses of a transform expression, each with their own semantics. The `copy` clause begins every transform expression and is comprised of one or more `"$" VarName ":=" ExprSingle` constructions that form a variable binding with `VarName` being the variable name, and `ExprSingle` being the source expression resulting in the data of the variable. The source expression must be a simple expression, else an `err:XUST0001` static error is raised. Moreover, this expression must evaluate to a single item otherwise an `err:XUTY0013` error is raised. Once evaluated, this resulting item is copied, including any items it may contain, meaning it is identical to the original item in every way except its identity. These copy variables can only be accessed within the scope of their transform expression, but not within the `copy` in which they were instantiated. Next comes the modify clause which contains one ExprSingle that must be an updating or vacuous expression, otherwise an err:XUST0002 static error is raised. The result of evaluating this expression is then a PUL, but the target items of the update primitives in the PUL must have been created in the previous copy clause – if not then an err:XUDY0014 dynamic error is raised. Finally, the return clause closes the transform expression, and it too contains one ExprSingle but this expression must be simple, otherwise an err:XUST0001 static error is raised. The resulting XDM instance from the return clause expression is then the result of the transform expression as a whole.

Syntax 2.2: Function Declarations

The general form of a function declaration is extended with the keyword "updating":

```
FunctionDecl ::= "declare" "updating"?  "function" QName "(" ParamList?  ")"
                   ( "as" SequenceType )?  ( EnclosedExpr | "external" )  (2.2)
```

Function Declarations allow for the creation of user-defined functions. Before the XQuery update facility, these functions could only be called as simple expressions, but with the "updating" keyword, the functions can be marked as returning a non-empty PUL, making calls to updating functions become updating expressions. The notion of function calls being updating or simple comes directly from the new semantics of the function declarations. If updating is not specified, then the function declaration must be a simple expression; otherwise an err:XUST0002 static error is raised. However, if updating is specified then the function declaration cannot include a return type without raising an err:XUST0028 static error. Moreover, the function declaration must be an updating expression or a vacuous expression; otherwise an error:XUST0002 static error is raised.

Beyond the extensions related to the updating keyword, the semantics of function declarations remain unchanged.

The semantics of several existing expressions are also extended, despite no change to their syntax, to help categorise them as simple, updating, or vacuous, and to set limitations on where an expression must be one of simple, updating, or vacuous.

FLWOR Expressions are the XQuery equivalent of SELECT, FROM, WHERE in SQL, allowing for the querying of collections of data. They are extended such that if a for, let, where, or order by clause contains an updating expression then an err:XUST0001 static error is raised. Moreover, the category of expression for the FLWOR expression is the same as the category of its return clause. Finally, if the FLWOR expression is updating then each tuple passing through the return clause will generate a PUL that is merged with the previously generated PULs using the upd:mergeUpdates routine.

Typeswitch Expressions are like a switch statement in C with case and default branches, except instead of switching on the value of an expression they switch on the type. They ensure that if the expression that is being "switched" upon is updating then an err:XUST0001 static error is raised. Each expression in any case or default branch can be simple, updating, or vacuous, but if any branch expression is updating, then all branch expressions must be updating or vacuous; otherwise an err:XUST0001 static error is raised. Thus, if any branch expression is updating, then so is the whole typeswitch expression. For the typeswitch to be vacuous, all branch expression must be vacuous. In all other cases the typeswitch is a simple expression. Lastly, if the typeswitch is updating then the resulting PUL comes directly from the matching updating branch.

Conditional Expressions allow for branching into different clauses based on a boolean condition and in XQuery they require both a `then` and an `else` clause to make up the if-then-else construct. They are made similar to typeswitch expressions in that the semantics depend on branches – `then` and `else` for conditional expressions. If the expression in the condition (`if` clause) is updating then an `err:XUST0001` static error is raised. Furthermore, one updating branch means both must be updating or vacuous; otherwise an `err:XUST0001` static error is raised. When all branches are vacuous the conditional expression is vacuous, but if at least one is updating then the conditional expression is updating. In any other case the conditional expression is simple. Finally, the PUL from an updating conditional expression comes from the updating branch that is selected.

Comma Expressions allow for expressions to be chained together. They are the crux of having a string of updating expressions create a large PUL acting like an updating transaction that you would normally find in a language like SQL. Once more, at least one updating expression in the comma expression means all others must be updating or vacuous, and so the comma expression is considered updating; otherwise an `err:XUST0001` static error is raised. If all expressions are vacuous then so is the comma expression. Any other case makes the comma expression simple. Then, to pool all the changes together in an updating comma expression, each expression generates a PUL which are merged together using the `upd:mergeUpdates` routine.

All of these changes enable XQuery to define a set of procedures that can modify the internal state of XDM instances to virtually any desired degree, all while maintaining consistency through proper isolation.

2.4 JSONiq

The focal query language for this thesis is JSONiq, a query and processing language specifically designed for the JSON data model and built on the shoulders of SQL and XQuery. XQuery is a similar query language but is designed for the XML data model instead. JSONiq borrowed several ideas from XQuery, namely the FLWOR construct, the functional paradigm, and declarative, snapshot-based updates. JSONiq then applies these concepts to JSON, a simpler data model than XML.

2.4.1 JSON

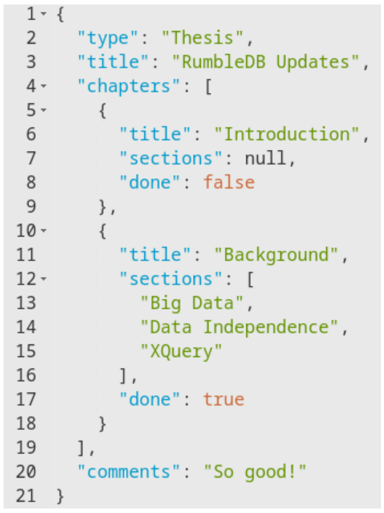
JavaScript Object Notation, JSON, is a textual data format used to represent semi-structured data of arbitrary nestedness (Crockford, 2006).

Four primitive types – strings, numbers, booleans, and nulls – comprise the atomic data of JSON, while two structured types – arrays and objects – enable nestedness; JSON values can be any of these types. Arrays are ordered sequences of zero or more values of any type, whereas objects are unordered collections of zero or more key-value pairs, in which the key is a string and the value can be of any type.

These constructions ensure that JSON documents can represent arbitrary nestedness and much heterogeneity, an example being figure 2.3. Having broken the atomic integrity and domain integrity constraints generally enforced by tabular formats, JSON lends itself to best representing tree-shaped data as depicted by figure 2.4.

2.4.2 JSONiq Data Model

JSONiq borrows much of its data model from XQuery, discussed above, and so it too treats a *Sequence* of *Items* as a first-class citizen. Moreover, JSONiq is set-oriented like SQL, but despite this, a sequence is ordered and allows for duplicates, more akin to *List* semantics; a sequence, unlike a List, is flat



```
{
  "type": "Thesis",
  "title": "RumbleDB Updates",
  "chapters": [
    {
      "title": "Introduction",
      "sections": null,
      "done": false
    },
    {
      "title": "Background",
      "sections": [
        "Big Data",
        "Data Independence",
        "XQuery"
      ],
      "done": true
    }
  ],
  "comments": "So good!"
}
```

Figure 2.3: Example of JSON object with nesting and heterogeneity

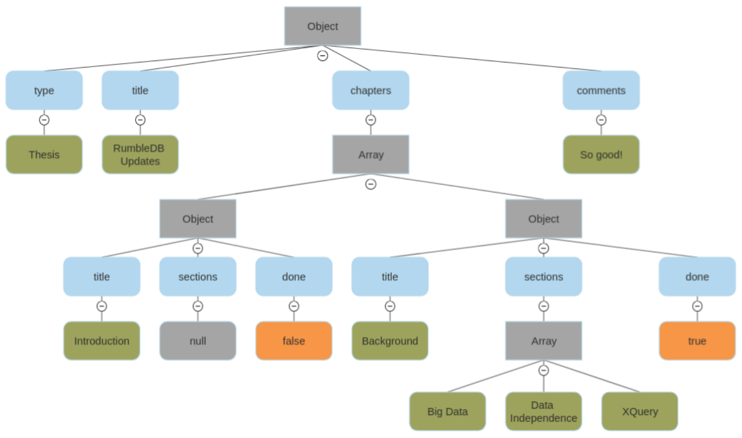


Figure 2.4: Tree structure representing the JSON in figure 2.3

and automatically unnested. Thus, a sequence cannot contain other sequences and if it does, these sequences are automatically unfurled and their items are placed into the outer sequence. Finally, sequences can be empty, and singleton sequences are considered the same as the item they contain.

Items on the other hand are much more heterogeneous as they represent all atomic and structured types available in standard JSON as well as many more, including time, date, duration, binary, URI, and more comprehensive number types. Additionally, items can have function types, matching JSONiq's functional paradigm, and user types in the form of annotated types. As such, JSONiq can handle the same heterogeneous, denormalised, semi-structured data format as represented by JSON, and more.

2.4.3 JSONiq Processing Model

Sequences being first-class citizens in JSONiq means that not only does every expression produce a sequence but they also all take sequences as input, thereby making JSONiq set-oriented in that JSONiq processes a sequence at a time as opposed to single data objects. The functional nature of JSONiq then asserts that every JSONiq program is an expression, itself composed of sub-expressions; every JSONiq building block is an expression. These programs are then declarative in that they only describe the shape and characteristics of the output, but cannot specify the implementation of the execution – instead relying on underlying optimisations to make these decisions based on the data being processed.

Combining these attributes with the extensive library of expressions available enables JSONiq queries to manipulate all the intricacies of arbitrarily nested, heterogeneous, tree-shaped data. The FLWOR (For, Let, Where, Order By, and Return) expression is the successor of SQL's `SELECT`, `FROM`, `WHERE` and is the core of JSONiq's queries. Instead of directly being comprised of expressions, a FLWOR expression is a chain of clauses in which each clause is composed of expressions. Through this chain of clauses stream *Tuples*, which are sets of variables bound to sequences that evaluate to a sequence of items once the stream meets the return clause. Using FLWOR expressions, JSONiq can facilitate the complex processing required to join, enrich, and clean heterogeneous datasets, to name a few of the possibilities.

2.4.4 JSONiq Updates

Similar to the XQuery Update Facility, (Robie et al., 2011) specifies the methodology behind updating objects, arrays and the items they contain in JSONiq. To do so, the processing model is extended such that every expression produces a sequence and a PUL, either or both of which may be empty, meaning that both cannot be non-empty. The definition of a PUL, the update routines that can act of them, and their snapshot semantics are largely the same as what is described in subsection 2.3.3, with the exception that the update primitives represent the change of state intended to be applied to a specified object or array. To account for this difference in the targets of update primitives, JSONiq introduces several update primitives, along with the expressions that generate them:

UP1 **jupd:insert-into-object($o as object(), $p as object())**: Inserts all key-values pairs of $p into $o.

UP2 **jupd:insert-into-array($a as array(), $i as xs:integer, $c as item()*)**: Inserts all items of the sequence $c into $a at position $i.

UP3 **jupd:delete-from-object($o as object(), $s as xs:string*)**: Removes all key-value pairs from $o whose key appears in the sequence of strings $s.

UP4 **jupd:delete-from-array($a as array(), $i as xs:integer)**: Removes the item at position $i in $a, while moving all following items down by one position.

UP5 **jupd:replace-in-object($o as object(), $n as xs:string, $v as item())**: Replaces the value of the key-value pair that has the key $n, in $o, with the item $v. If no such key $n is present in $o, then nothing occurs.

UP6 **jupd:replace-in-array($a as array(), $i as xs:integer, $v as item())**: Replaces the value located at position $i in $a with the item $v. If $i is outside of the range 1 to jdm:size($a) inclusive, then nothing occurs.

UP7 **jupd:rename-in-object($o as object(), $n as xs:string, $p as xs:string)**: Replaces the key of the key-value pair that has key $n, in $o, with the key $p. If no such key $n is present in $o, then nothing occurs.

In order to generate update primitives and modify items, six expressions are introduced, and the semantics of functions are extended. However, only the semantics of delete, insert, replace, rename, and append expressions are specified by (Robie et al., 2011) with any additional semantics being assumed from the XQuery Update Facility of subsection 2.3.3.

Syntax 2.3: Delete Expressions

The general form of a delete expression is:

```
JSONDeleteExpr ::= "delete" "json" PrimaryExpr ( "(" ExprSingle ")" )+   (2.3)
```

Delete Expressions enable the removal of key-value pairs from objects or items from an array through the use of the `jupd:delete-from-object` and `jupd:delete-from-array` update primitives. The `PrimaryExpr` acts as our base expression from which all but the last `ExprSingle` are evaluated as dynamic function calls (array or object look-ups) to eventually return a single array or object, otherwise error `jerr:JNUP0008` is raised. The final `ExprSingle` is then either an array look-up on array `$a` at index `$i`, or an object look-up on object `$o` with string `$s`. In the case of an array look-up (delete expression of form `delete json $a[[$i]]`), `$i` is cast to an `xs:integer` and a `jupd:delete-from-array($a, $i)` update primitive is created. Failing to cast `$i` to an `xs:integer` raises a `jerr:JNUP0007` error, and `$i` outside of the index range of `$a` raises a `jerr:JNUP0016` error. Whereas, in the case of an object look-up (delete expression of form `delete json $o.$s`), `$s` is cast to an `xs:string` and a `jupd:delete-from-object($o, $s)` update primitive is created. Failing to cast `$s` to an `xs:string` raises a `jerr:JNUP0007` error, and a `$s` that is not a key of `$o` raises a `jerr:JNUP0016` error.

Syntax 2.4: Insert Expressions

Insert expressions have two distinct forms

```
JSONInsertExpr::= "insert" "json" ExprSingle "into"
                  ExprSingle ("at" "position" ExprSingle)?          (2.4)
```

and

```
JSONInsertExpr ::= "insert" "json" PairConstructor ("," Pair Constructor)*
                                              "into" ExprSingle
                                                                    (2.5)
```

Insert Expressions allow for the insertion of key-value pairs into objects or of items into an array with the `jupd:insert-into-object` and `jupd:insert-into-array` update primitives. The latter of the possible syntaxes can be interpreted as a specific case of the former where the comma separated `PairConstructor` values, of the latter, define an object in the place of the first `ExprSingle` in the former. As such,
`insert json "foo" : "bar", "bar" : "foo" into $o`
and
`insert json { "foo" : "bar", "bar" : "foo" } into $o`
are equivalent and we can focus on the first of the two forms of insert expressions. The second `ExprSingle` will evaluate to either an array or an object, correspondingly the insert expression will be an array-insertion or an object-insertion.

An array-insertion expression looks like `insert json $c into $a at position $i`, where `$a` is the array to insert into, `$i` is the index to insert into, and `$c` is the sequence of items to insert. When evaluating an array-insertion expression, `$i` is cast to an `xs:integer`, all items in `$c` are copied, and a `jupd:insert-into-array($a, $i, $c)` update primitive is created. If `$a` is not an array then a `jerr:JNUP0008` error is raised, and if `$i` cannot be cast to an `xs:integer` then a `jerr:JNUP0007` error is raised.

An object-insertion expression has a simpler syntax and takes shape `insert json $p into $o`, where `$o` is the object to insert into, and `$p` is a sequence of objects containing all key-value pairs to insert into `$o`. Evaluating an object-insertion expression involves converting `$p` into a single object and copying the result, then creating a `jupd:insert-into-object($o, $p)` update primitive. If `$o` is not an object then a `jerr:JNUP0008` error is raised, if `$p` is not a sequence of objects then a `jerr:JNUP0019` error is raised, and if converting `$p` into a single object causes any pair collisions – i.e. more than one object of `$p` share a key – then a `jerr:JNDY0003` error is raised.

Syntax 2.5: Rename Expressions

Rename expressions take the shape

```
JSONRenameExpr ::= "rename" "json" PrimaryExpr ( "(" ExprSingle ")" )+
                                                          "as" ExprSingle  (2.6)
```

Rename Expressions permit the renaming of keys in the of key-value pairs of objects by using the `jupd:rename-in-object` update primitive.

As in delete expressions, the `PrimaryExpr` followed by all but the last `ExprSingle` in the `( "(" ExprSingle ")" )+` construction, evaluate as dynamic function calls to return a single array or object. However, since renaming can only alter the keys of object, in rename expressions only a single object is expected and a `jerr:JNUP0008` error is raised otherwise.
All rename expressions then simplify to the form `rename json $o.$s as $n`, where `$o` is the object with a key to rename, `$s` is the key to rename, and `$n` is the new name of the key. To evaluate these, `$s` is cast to an `xs:string` and a `jupd:rename-in-object($o, $s, $n)` update primitive is created. If the cast of `$s` fails then a `jerr:JNUP0007` error is raised, and if `$o` does not contain a pair with key `$s` then a `jerr:JNUP0016` error is raised.

Syntax 2.6: Replace Expressions

Replace expressions have the form

```
JSONReplaceExpr ::= "replace" "json" "value" "of"
                     PrimaryExpr ( "(" ExprSingle ")" )+ "with" ExprSingle  (2.7)
```

Replace Expressions enable the replacing of the values of key-value pairs in objects or items at specific indexes in an array by using the `jupd:replace-in-object` and `jupd:replace-in-array` update primitives. Once more, the `PrimaryExpr ( "(" ExprSingle ")" )+` construction is evaluated with dynamic function call semantics and must return a single object or array, otherwise a `jerr:JNUP0008` error is raised. From here, interpreting the last `ExprSingle` creates an array-lookup case and an object-lookup case as seen in delete expressions.

The array-lookup case occurs when the replace expression has the pattern `replace json value of $a[[$i]] with $c`, where `$a` is an array, `$i` is cast to an `xs:integer`, and the sequence resulting from `$c` is processed into a null item when the sequence is empty, a single item when the sequence contains only said item, and an array of the items in the sequence when the sequence contains more than one item. The single item result of `$c` is then copied and the `jupd:replace-in-array($a, $i, $c)` update primitive is created. Failing to cast `$i` raises a `jerr:JNUP0007` error, and if the cast succeeds but `$i` is outside of the index range of `$a` then a `jerr:JNUP0016` error is raised.

Whereas the object-lookup case occurs when the replace expression has the pattern `replace json value of $o.$s with $c`, where `$o` is an object, `$s` is cast to an `xs:string`, and the sequence resulting from `$c` is processed into a single item – as in the array-lookup case – which is copied and the `jupd:replace-in-object($o, $s, $c)` update primitive is created. Failing to cast `$s` raises a `jerr:JNUP0007` error, and if the cast succeeds but `$s` is not a key of a pair in `$o` then a `jerr:JNUP0016` error is raised.

Syntax 2.7: Append Expressions

Append expressions take the form

```
JSONAppendExpr ::= "append" "json" ExprSingle "into" ExprSingle  (2.8)
```

Append Expressions are similar to insert expressions, but only allow for the insertion of items at the end of an array with the `jupd:insert-into-array` update primitive.

The semantics of an append expression are nearly identical to those of the array-insertion case in an insert expression, with the omission of the `"at" "position" ExprSingle` construction as append expression insert items into the end of an array. Thus, when executing an append expression with the pattern `append json $c into $a` – where `$a` is an array and `$c` is the sequence of items to append – only a `jupd:insert-into-array($a, len($a) + 1, $c)` update primitive is created with the insertion index being the length of the array plus one. If `$a` is not an array, then a `jerr:JNUP0008` error is raised.

Update Routines

Basic-updating expressions require integration with more complex updating expressions, and special expressions, namely the transform expression. Each basic-updating expression generates a single update primitive which is stored in a PUL, but certain updating expressions, like FLWOR expressions, can invoke the execution of several basic-updating expressions which in turn produce several PULs. Since each expression can only return a single PUL, these more complex updating expressions must use `upd:mergeUpdates` to compress their PULs into one. Moreover, the changes represented by PULs need to be enacted either at the end of a query or after the `modify` clause of a transform expression to maintain their snapshot consistency; the use of `upd:applyUpdates` does exactly this. Nevertheless, both `upd:mergeUpdates`, and `upd:applyUpdates` have semantics in JSONiq that differ from their xQuery counterparts and so require clarifying.

Function Signature 2.1: Merging Updates

The update routine `upd:mergeUpdates` takes in two PULs, merges their update primitives, and returns a single PUL to give the following function signature, in Java syntax

`PUL mergeUpdates(PUL pul1, PUL pul2)` (2.9)

upd:mergeUpdates is adapted to account for the targets of update primitives being either an object or an array, and so has a unique merging action for each update primitive. First considering objects, their semantics must be maintained while the updates are merged. For object insertions (UP1) this means that all keys to insert into a single object should be collated in a single, new UP1, but if one key to insert is shared by multiple UP1s then this should flag as a – would-be – key collision and throw a `jerr:JNUP0005`. Whereas more leniency is available for object deletions (UP3) in that, similarly, all keys to delete from a single object targeted by many UP3s are collected to create a single, new UP3, but key collisions are not a problem in this case as any duplicates are ignored instead. Even simpler are the object replacements (UP5) and object renamings (UP7) as no replacement nor renaming on the same key within the same object can take precedence over another within the same PUL as they are in the same snapshot, and so replacements of the same key in the same object raise a `jerr:JNUP0009` error, while renamings of the same key in the same object raise a `jerr:JNUP0010` error. Lastly, for objects, an UP3 makes redundant any UP5 or UP7 that targets the same object and key since at the end of the snapshot the key would not remain to show the changes of UP5s or UP7s; hence these UP5s and UP7s can be ignored.

Regarding update primitives that target arrays, the semantics are similar to those for objects with the exception that an integer index is to be accounted for instead of a string key, and so an ordering is present. Unlike UP1, array insertions (UP2) do not need to worry about collisions during merging as UP2s acting on the same array at the same position are merged into a single, new UP2 with a sequence of all the items to insert from the previous UP2s – crucially, the order of this sequence is implementation-defined and so any ordering can be valid. Array deletions (UP4), however, are akin to UP3 in that multiple UP4s on the same array at the same index simply create a single, new UP4 and remove duplicates. Likewise, array replacements (UP6) are treated analogously to their object counterparts, and so more than one UP6 on the same array at the same index raises a `jerr:JNUP0009` error, for the simple reason that no replacement can take precedence over another within the same snapshot. Finally, for the same reasoning as given above for objects, UP4 make any UP6, on the same array and index, obsolete and so any such UP6s are ignored.

Once all varieties of update primitives have been merged and pruned, they can be collected in a resultant PUL and provided as the return value of the `upd:mergeUpdates` routine. Figure 2.5 adds visual clarification to the explanations above.

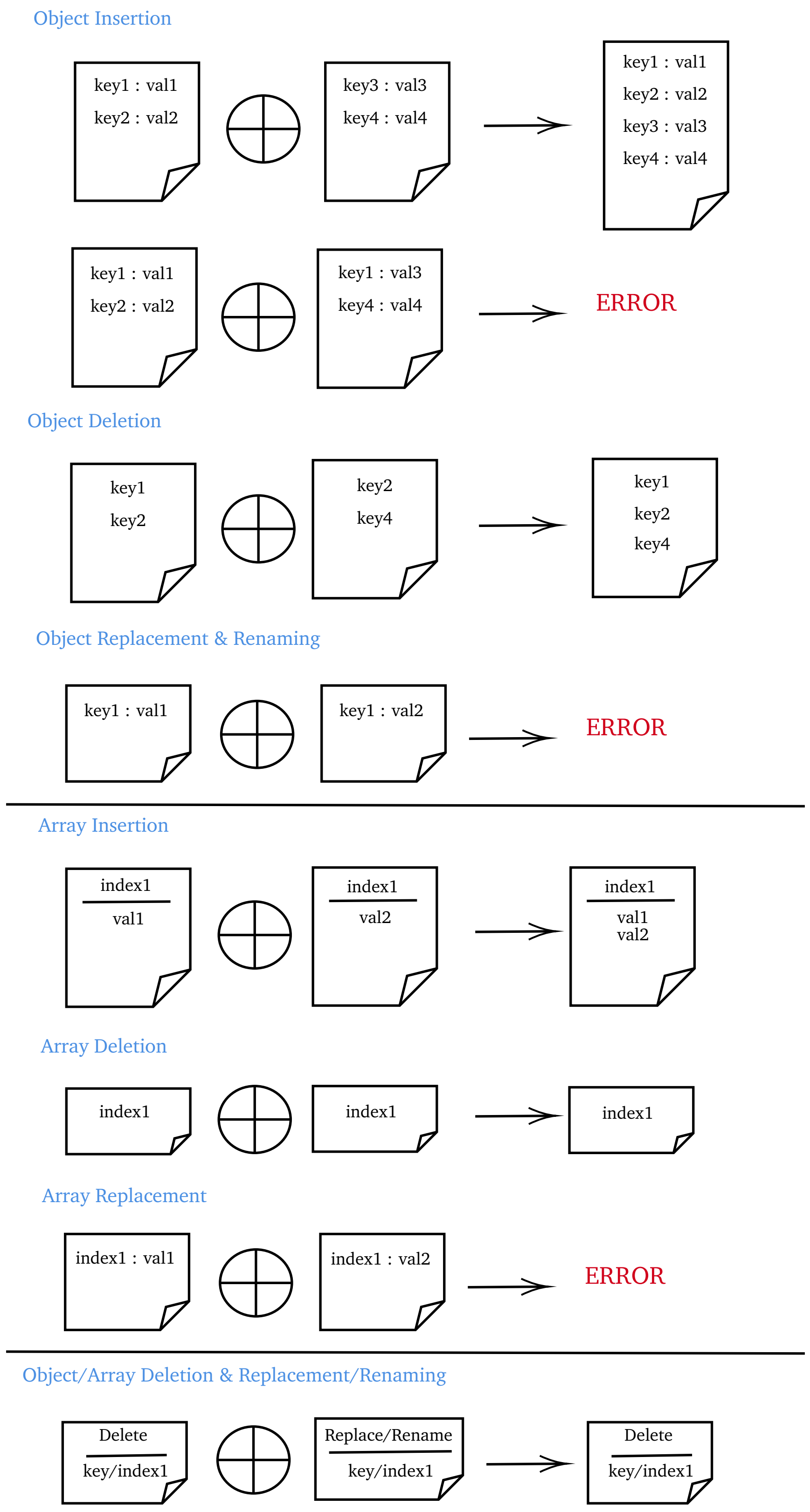


Figure 2.5: Visualisation of merging operations as applied to the specified update primitives

Function Signature 2.2: Applying Updates

The update routine `upd:applyUpdates` takes in a single PUL, applies the changes described by its update primitives and returns nothing to give the following function signature, in Java syntax

`void applyUpdates(PUL pul)` (2.10)

upd:applyUpdates sees very few changes as it still applies the state changes described by the update primitives of the PUL, and signifies the end of a snapshot. However, before any update can be applied, the array or object that is being targeted for the update is “locked onto” by resolving the expression that should evaluate to either an integer index, for arrays, or a string key, for objects. Nevertheless, these expressions can result in an empty sequence, in which case the update primitive is ignored as no change is specified. Naturally, this preliminary “locking on” does not occur for UP1 since the key is not yet present in the target object and so no evaluation against the target can occur.

Once all targets have been identified and any selector has been “locked in” then the update primitive can be applied as described at the start of this section. Importantly, the order of application should not matter since each update primitive should act independently of one another and so no conflict can occur to disrupt the data model. After all the update primitives have been applied the current snapshot ends, meaning future PULs will exist under a snapshot that contains the changes applied during this update routine.

Adapting the semantics presented by the XQuery Update Facility to JSON allows JSONiq to update any structured item to an arbitrary degree of nestedness, while maintaining the consistency and isolation available in XQuery. Combining the power of the XQuery Update Facility with the simplicity of JSON stands to cement JSONiq as a potent query processing language for highly nested and heterogeneous data, now capable of handling fully transactional processing.

2.5 RumbleDB

RumbleDB, (Müller et al., 2020), is a query execution engine for large, messy datasets that implements the JSONiq query language discussed in section 2.4. Much like JSONiq, RumbleDB is built on the shoulders of giants and so does not aim to outright replace any current big data technologies, but rather piece them together to create a whole greater than the sum of its parts; hence, RumbleDB is written in Java, allowing it to take full advantage of any Big Data APIs needed to meet its goal. The primary goal of RumbleDB is to be an implementation of and provide data independence for the heterogeneous, semi-structured data found in Big Data workflows. Thus, the user needs only to interface with the JSONiq data model despite the numerous modes of execution RumbleDB has at its disposal.

2.5.1 Execution Modes

RumbleDB employs three different execution modes to adjust to user query requirements while optimising processing time. These modes are dubbed *Local*, *RDD* and *Dataframe*. The largest distinction between these modes is their ability to parallelise; only RDD and Dataframe executions are able to parallelise. As their names suggest, the RDD and Dataframe modes rely on the Spark/SparkSQL Java API, whereas the local execution uses RumbleDB’s implementation of an open-next-close iterator processing pattern (Graefe, 1993) over the JSONiq data model. Regardless of the underlying execution mode, every execution produces a JSONiq sequence, meaning the user need only interact with JSONiq, while taking full advantage of a variety of processing models. This separation of user query writing and execution mode is then the heart of RumbleDB's data independence.

2.5.2 From Query to Result

Execution modes only describe the processing model used for a given query, so to actually use these execution modes on a JSONiq query RumbleDB must make several transitions: from the JSONiq query to an abstract syntax tree (AST); from the AST to a JSONiq expression tree; from a JSONiq expression tree to a JSONiq runtime iterator tree, holding the execution implementations; finally, from a JSONiq runtime iterator tree to a JSONiq sequence.

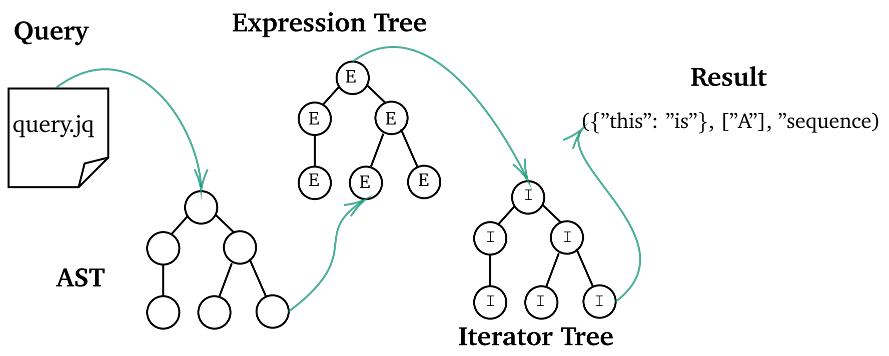


Figure 2.6: Flow of RumbleDB processing from a JSONiq query to the resulting sequence

2.5.3 AST Generation

When a user writes a JSONiq query, RumbleDB must lex and parse the text and generate the tree of JSONiq expressions that compose the query. Given that this task is common throughout programming language compilations, many technologies exist to perform the lexing and parsing of programs – all they need is a grammar to work against. RumbleDB employs ANTLR4 (Parr, 2013) for the lexing and parsing of JSONiq queries. Thankfully, the JSONiq specification (Robie et al., 2011) was designed meticulously and the grammar described can be used directly for RumbleDB's parsing. ANTLR4 in conjunction with the JSONiq grammar creates an AST from which RumbleDB can generate an expression tree.

2.5.4 Expression Tree Generation

JSONiq is a functional language meaning that each query can be broken down into component expressions that are nested or chained together in some fashion. Thus, generating an expression tree to describe the information demanded by a query can be done using the AST. RumbleDB applies several visitor patterns (Palsberg and Jay, 1998) to translate the AST to a JSONiq expression tree and then populate this expression tree with data relevant for later processing.

Each expression is represented by a class that `extends` the `Expression` class, which has protected member variables constituting information relevant to each expression, such as the type of sequence – empty, single item, etc. – returned by this expression. These sub-classes then have their own member variables to accommodate the expressions that comprise them, like the expressions for the if-condition, then branch, and else branch of a conditional expression. This nesting of expressions forms an expression tree to represent the logical structure of the query which can be used to create a runtime iterator tree.

2.5.5 Iterator Tree Generation

In contrast to the logical structure described by the expression tree, the runtime iterator tree describes the flow of execution for a given query. For RumbleDB, the runtime iterator tree is the equivalent of the executable made by a compiler and so it outlines the methods for executing the query, guided by the execution modes of subsection 2.5.1. In order to create the runtime iterator tree from the intermediary expression tree, another set of visitors is used to instantiate and populate the runtime iterator classes.

The `RuntimeIteratorInterface` interface and `RuntimeIterator` abstract class create a template for the information and methods a runtime iterator must provide in order to be executable under RumbleDB's execution modes. An almost one-to-one mapping can be created between the expression classes and the runtime iterator classes in that almost every expression has a corresponding runtime iterator and any nested expressions from the expression class are reflected in the runtime iterator class, once more creating a tree structure. The distinctions arise when looking at expressions like function calls which can correspond to many functions, such as `Max` and `Min`, each requiring their own method of execution. As each runtime iterator `extends` the `RuntimeIterator` class they must each provide an implementation for at least local execution, if not also RDD, and Dataframe execution.

Local Execution API

Local execution is provided through an API in the `RuntimeIterator` class, consisting of: `void open()`, to prepare the iterator for execution; `boolean hasNext()`, to indicate whether any more results can be produced; `Item next()`, to materialise the result in the form of an `Item` interface to be discussed later; and `void close()`, to stop the iterator from processing anything more. Each of these methods will recurse down the tree, accessing the same API of other iterators to ensure all aspects of the query are accounted for.

As the name implies, local execution is built for processing on a single machine, hence the use of the Volcano processing to process one item at a time and minimise memory usage.

RDD Execution API

RDD execution has a simple API split across the `RuntimeIterator` and `HybridRuntimeIterator` abstract classes that differentiate between the Spark APIs for RDDs and Dataframes. As described in (Zaharia et al., 2016), RDDs, or Resilient Distributed Datasets, are the primary building block of the Spark API and are simply immutable, distributed sets of data objects without any schemas. Whereas, Dataframes are RDDs of records with a known schema, meaning that any parallel execution done using Dataframes can also be completed using RDDs, and RumbleDB is aware of this fact. Thus, the `RuntimeIterator` class has a `JavaRDD<Item> getRDD()` method to materialise the resulting sequence of items from parallel processing as a `JavaRDD<Item>`, and a `boolean isRDDOrDataframe()` method to signify if the current execution mode is for RDDs or Dataframes, i.e. execution is not local. The `HybridRuntimeIterator` then overrides the `JavaRDD<Item> getRDD()` method to use a new method when `isRDDOrDataFrame()` returns `true` but the execution mode is not for Dataframes – `JavaRDD<Item> getRDDAux()` creates the `JavaRDD<Item>` using RDD transformations from the JavaSpark API. `getRDDAux()` is abstract in this class and so is implemented by the concrete classes that `extend HybridRuntimeIterator`, which may call `getRDDAux` on iterators that are their direct descendants in the tree, thereby creating a chain of calls following a depth-first search approach.

Dataframe Execution API

Dataframe execution has an API analogous to that of RDDs with the `RuntimeIterator` class specifying a `boolean isDataFrame()` method to signify if the current execution mode is for Dataframes, and a `JSoundDataFrame getDataFrame()` method to materialise the resulting DataFrame, often using SparkSQL queries and temporary views. In order to adapt Spark Dataframes to JSONiq, RumbleDB uses the `JSoundDataFrame` class as a wrapper for Dataframes (of type `Dataset<Row>` in Java) that also contains an `ItemType` acting as the JSONiq representation of the Dataframe's schema. These `getDataFrame()` methods are then implemented for concrete classes and chain calls with the same depth-first search structure described for RDD execution.

This array of `RuntimeIterator` classes and its subclasses then form the iterator tree and specify the possible flows of execution ready to generate JSONiq sequences.

2.5.6 Sequence Generation

The final step for RumbleDB to create the result of a query is to use the runtime iterator tree to generate a JSONiq sequence of items.

JSONiq items are implemented in RumbleDB via the `Item` interface which provides dummy `default` methods for every JSONiq item type. Within this interface, each item has methods to retrieve or access the encapsulated Java value – like `boolean getBooleanValue()` for JSONiq boolean items, and `Item getItemByKey(String key)` for JSONiq objects – and each item has methods to verify its type – like `boolean isArray()` returning `true` only for JSONiq arrays – thereby maintaining the JSONiq data model as all items need only be accessed via the `Item` interface and not the implemented classes.

JSONiq sequences are implemented using the `SequenceOfItems` class that acts like a wrapper for the `RuntimeIterator` class, ensuring that it can only be used to create a `List<Item>` object that represents the JSONiq sequence.

Generating a `List<Item>` requires the `RuntimeIterator` class to provide several methods to "materialize" the sequence of items, such as `Item materializeExactlyOneItem()`, but these are all just variations of the `List<Item> materialize()` method which follows one of the execution flows, described in subsection 2.5.5, based on the iterator's execution mode.

Local execution necessitates an iterator model for materialisation, as outlined by code block 2.1.

Code 2.1: Local Materialisation

Materialisation of a sequence from a locally executed iterator has the following iterator style:

```
List<Item> result = new ArrayList<>();
iterator.open();
while(iterator.hasNext()) {
    result.add(iterator.next());
}
iterator.close();
return result;
```

RDD and Dataframe execution have similar materialisation flows, with the exception that the Dataframe materialisation must account for a schema. Code block 2.2 highlights this difference.

Code 2.2: RDD & Dataframe Materialisation

As discussed, RDDs and Dataframes both use the `getRDD()` method as a `JavaRDD<Item>` can be materialised by calling the `collect()` action on an RDD like so:

```
JavaRDD<Item> items = iterator.getRDD();
List<Item> collectedItems = items.collect();
return collectedItems;
```

Since materialisation can be done directly on `JavaRDD`s, deriving the resultant RDD to collect is simple for the RDD execution mode (as `getRDDAux()` can be used) but the `getDataFrame()` method returns a `JSoundDataFrame` that must be converted to a `JavaRDD<Item>`. Therefore the `JavaRDD<Item> dataFrameToRDDOfItems(JSoundDataFrame df)` method is used to map each row of the DataFrame to a JSONiq item by comparing the `JSoundDataFrame`'s `ItemType` to the schema of the Spark DataFrame. A `getRDD()` method could then resemble:

```
JavaRDD<Item> getRDD() {
    if (iterator.isDataFrame()) {
        JSoundDataFrame df = iterator.getDataFrame();
        return dataFrameToRDDOfItems(df);
    } else {
        return iterator.getRDDAux();
    }
}
```

During the materialisation, however, complications arise when iterators that are part of the same iterator tree do not support the same execution modes. The most common example occurs when iterators executing locally receive RDD or Dataframe inputs from child iterators. Nevertheless, this problem is overcome by interlacing the different forms of materialisation, depicted in code blocks 2.1 and 2.2, depending on the required execution mode.

Finally, the generated `SequenceOfItems` can be accessed with a `long populateList(List<Item> resultList)` method to materialise the concrete sequence of JSONiq items resulting from the query.

2.6 Persisting Updates

Providing the ability to alter data objects, instead of completely overwriting them, is just the first challenge of facilitating updates. The next challenge comes when actually persisting these changes beyond the process in which the updates were created. Thus far, subsection 2.4.1 and 2.4.2 have described the JSON data format and how to model it, while subsection 2.4.4 outlines a specification for updating JSON data in place, and section 2.5 shows an implementation of JSONiq capable of housing such a specification. The only challenge left to overcome is to discover how to store, and persist these updates.

2.6.1 Transactions

User interactions with a database generally consist of operations that read data from the database – read operations – and operations that write data to the database – write operations. A transaction is a set of read and/or write operations on a database that follow the ACID principles, as outlined by (Haerder and Reuter, 1983), although the most interesting cases occur when the goal of transactions is to modify the data.

ACID principles, beyond being a lovely acronym, can be summarised as:

1. **Atomicity:** all operations in a transaction must occur in their entirety, or none of them at all
2. **Consistency:** transactions must transition the database from one stable state, defined by the database system, to another
3. **Isolation:** concurrent transactions are independent of one another, and so produce predictable results
4. **Durability:** modifications to state made by completed transactions will persist beyond system failure

Virtually all relational data management systems (RDBMS), from PostgreSQL (Group) to MySQL (mys), are ACID-compliant, and even some NoSQL solutions, like Apache CouchDB (Anderson et al., 2010), are too. Thus, it seems that, thanks to their durability, any ACID-compliant system should be perfectly capable of persisting updates to JSON via RumbleDB. However, RumbleDB is built for the Big Data paradigm where JSON is extremely popular – with JSON being used as a primary format for large data collections like the GitHub archive (gha) – and notoriously, Big Data systems struggle to be ACID-compliant (Moniruzzaman and Hossain, 2013). The disparity between Big Data and ACID is discussed, and ACID is compared with BASE as an alternative in (Banothu et al., 2016). However, BASE was built with inspiration from the CAP theorem.

CAP is a theorem, thought up in (Brewer, 2000), to better describe the trade-offs that different data systems make to meet their requirements. CAP is well outlined by (Gilbert and Lynch, 2012) and stands for:

1. **Consistency:** differing between systems, consistency can refer to the consistency in ACID for relational systems, but is broadly thought of as each server in a system returning the correct response upon request
2. **Availability:** each system request eventually receives a response
3. **Partition Tolerance:** the system of nodes continues to function despite being split into multiple independent groups due to some network failure impeding communications

The theorem states that any distributed system can have at most two of the properties at the same time, and herein lies the trade-off. ACID-compliant RDBMS are examples of traditionally single-site databases that only support the C and A properties of CAP, as they are generally centralised.

BASE as a paradigm, described by (Pritchett, 2008), looks to account for the pitfalls of ACID in distributed systems by noting that these systems demand reliability. In order to be ACID-compliant, data systems have to provide many features and add compute power that is inefficient for distributed systems as they impede their reliability. Thus, the BASE principles, cleverly mirroring its alternative in name, are as follows:

1. **Basically Available:** a system will always return a response to every request, even if the response is a notification of failure
2. **Soft State:** the system state is malleable, to mean that there is always the possibility to be changing, even without user access, in order to maintain eventual consistency
3. **Eventual Consistency:** given enough time without transactions, changes to the state will propagate to all nodes in the system and thus the system will become consistent; however, no guarantee is made about the consistency immediately after a transaction

In relation to the CAP theorem, BASE looks to provide A and P, with the hope that C comes eventually. Thus, BASE is a popular set of principles for Big Data systems where the volume of data means consistency is less important, and the need to house these large volumes requires distributed storage systems that must be available and able to partition to ensure they can be used everywhere and all the time.

Using ACID, BASE, and CAP allows for well-reasoned discernments about the best technologies for persisting updates in a Big Data system.

2.6.2 Delta Lakes (Databricks)

Standard data lakes, mentioned in section 2.1, often use object stores, like S3 (Ama, 2002), which prioritise a more BASE-like paradigm to maintain the A and P aspects of CAP since these stores tend to be eventually consistent (ama, 2002). However, the ideal solution would be to integrate more consistency into these stores so that they also receive ACID benefits, which delta lakes aim to provide. Introduced in (Armbrust et al., 2020), delta lakes extend data lakes with an ACID-compliant table storage layer. To achieve some ACID compliance, delta lakes maintain information about which objects are on a delta table in an ACID manner – with a write-ahead log also stored in an object store. This log is also used for other metadata information like per-file min-max statistics for faster searches. Alongside the log, delta tables encode the data objects in Parquet, a column-oriented data format with mandatory schemas (apa, c). With this configuration, delta tables are able to use spark-SQL to describe updating transactions that can be snapshot, logged, and applied to data objects in an ACID manner.

Delta lakes allow for:

1. **UPSERTs:** the update of values of a row it has a primary key that already exists and the insertion of new rows with primary keys that do not exist
2. **DELETEs:** the removal of a row from a table
3. **MERGEs:** the insertion of distinct ("non-matching") rows from two target tables into a new table, and the update of matching rows for the new table

4. **Schema Evolution:** change the schema of the table (e.g. by inserting a column) while still accessing older parquet files, for the same table, that lack the new schema

Hence, delta lakes provide at least the basic updating operations of SQL, but for more heterogeneous, nested schemas similar to that of the JSONiq data model in subsection 2.4.2.

Discussed in section 2.5, RumbleDB relies on the Apache Spark API to meet the processing needs of Big Data querying. Fortuitously, delta lakes too – being a Databricks product – have Spark API integration as a primary feature, thereby allowing for easier integration of delta lakes into RumbleDB.

Delta lakes are able to accommodate all of the requirements outlined at the beginning of this section and are therefore the focal technology to persist updates for this thesis.

3 Implementation of Updates

The primary aim of this thesis is the extension of RumbleDB with the JSONiq updates specification (2.4.4) to allow for the update of deeply nested and heterogeneous JSON fields in an intuitive and versatile manner. To achieve this, one requires extending all the components of RumbleDB that take a JSONiq script from query to result (2.5.2) with respect to both the XQuery Update Facility 1.0 (2.3.3) and the JSONiq update specification (2.4.4). Since we are primarily concerned with the implementation of updates for JSONiq, to ensure correctness we use local execution as the execution mode for the initial work before considering more parallelised executions.

The secondary aim of this thesis is to allow for the persistence of any updates made via RumbleDB. Thus, we extend RumbleDB to be compatible with Delta Lake – see subsection 2.6.2 – enabling users to query, update, and store their heterogeneous data, all via RumbleDB. In order to accomplish this integration, we only extend the iterator logic of RumbleDB since updates to Delta Lakes still interface through JSONiq expressions, a benefit provided by RumbleDB's data independence. Moreover, unlike implementing the conventional JSONiq updates, persisting updates to Delta Lakes necessitates the use of the Dataframe execution mode to access the SparkSQL API that RumbleDB and Delta Lakes share. Therefore, we partially extend functionality for both local and Dataframe executions.

3.1 Grammar Extension

The first step to actualising JSONiq updates in RumbleDB is to add the textual representation of the queries to the grammar. As mentioned in subsection 2.5.3, RumbleDB uses ANTLR4 to handle the lexing and parsing of the JSONiq grammar. The grammar is stored in a grammar file called `Jsoniq.g4` which ANTLR4 then uses to generate the token file, the lexer class, the parser class, and the visitor class – used to generate the Abstract Syntax Tree (AST) – for JSONiq. Thus, the grammar file directly influences the creation of the AST and what information is available in the AST to be used for creating the Expression Tree. A grammar file must then be well made (to aid later development) and above all, correct.

To extend the grammar file, we introduce adaptations of syntaxes 2.1 to 2.8. Here we now briefly explain some terms specific to grammars (Grune and Jacobs, 2006): *terminal symbols* are the non-divisible symbols that make up a grammar, while *non-terminal symbols* are symbols that can be replaced by groups of terminal symbols as defined by a set of production rules described in the grammar file. Moreover, ANTLR4 allows for the assignment of variables to non-terminal symbols, as seen first in 3.1, meaning these symbols can be easily referenced later when building the expression tree from the AST.

3.1.1 Transform Syntax

Starting with the Transform Expression (2.1), we introduce a new non-terminal symbol to simplify the declaration of variables to be copied for the transform expression. This non-terminal symbol is named `copyDecl`.

Syntax 3.1: Copy Declarations

Copy declarations for Transform Expressions take the form:

```
copyDecl :  var_ref=varRef ':=' ExprSingle
                                          varRef ::= '$' varName   (3.1)
```

Where `varRef` is an already present non-terminal representing a reference to a variable – a $ followed by a variable name, as above.

Using the `copyDecl` non-terminal we can simplify the Transform Expression's production rule.

Syntax 3.2: RumbleDB Transform Expressions

Transform Expressions in RumbleDB take the form:

```
transformExpr :  'copy' copyDecl ( ',' copyDecl )*
               'modify' mod_expr=exprSingle 'return' ret_expr=exprSingle  (3.2)
```

Syntax 3.2 concisely describes the shape of a transform expression, while taking advantage of the `copyDecl` non-terminal in conjunction with ANTLR4's variable assignment to ensure that the expression is compartmentalised into each copied variable and each clause. This division allows for easy pattern matching between the components of the syntax and the resulting transform expression, as part of the expression tree. Moreover, it ensures that copy declarations are considered separately from the transform expression which will prove useful for scoping. The treatment of these copy declarations is then akin to the treatment of ordinary variable declarations.

3.1.2 Function Declaration Syntax

The implementation of RumbleDB function declarations depicted in Syntax 3.3 experiences little to no changes in the syntax relative to Syntax 2.2. We only extend the function declaration by adding a flag indicating if the function is updating or not. Adding the flag to the grammar is simple and adds a boolean value to the AST.

Syntax 3.3: RumbleDB Function Declarations

Function declarations are extended with an optional `'updating'` flag, assigned to an ANTLR4 AST variable `is_updating`:

```
functionDecl :  'declare' (is_updating='updating')?  'function' fn_name=qname
            '(' paramList?  ')'  ('as' return_type=sequenceType)?
                    ('' (fn_body=expr)?  ''  | is_external='external')  (3.3)
```

By adding the `is_updating` variable, we can store information as to whether the function is updating in the AST for later processing.

3.1.3 Update Locator Syntax

Many of the expressions, namely Delete, Rename, and Replace expressions, have a common pattern. This pattern is used to identify the target expression (object or array) of the update and then access either the key or index that is to be updated. To simplify the production rules for these updating expressions, we introduce the `updateLocator` non-terminal.

Syntax 3.4: Update Locators

Below are two forms of the `updateLocator` production rule, where 3.4 shows the rule in alignment with the JSONiq specification of section 2.4.4

```
updateLocator :  PrimaryExpr ( "(" ExprSingle ")" )+          (3.4)
```

and 3.5 shows the rule used in RumbleDB which makes a distinction between an array lookup (e.g. `arr[[1]]`) and an object lookup (e.g. `obj.key`)

```
updateLocator :  main_expr=primaryExpr ( arrayLookup | objectLookup )+   (3.5)
```

The non-terminal 3.5 proves useful as it extracts the lookup syntax shared by several expressions, and allows the resolution of this lookup to be implemented independent of the updating expression. Moreover, it will make further updating expressions more concise.

3.1.4 Delete Syntax

The syntax of delete expressions is greatly simplified from the syntax of 2.3 to 3.6 in large part due to the use of 3.5. Moreover, the `'json'` syntax is removed to avoid bloating the grammar with redundancies. This removal is common throughout all expressions referenced in section 2.4.4.

Syntax 3.5: RumbleDB Delete Expressions

Delete Expressions in RumbleDB take the shape:

```
deleteExpr :  'delete' updateLocator  (3.6)
```

3.1.5 Replace Syntax

Replace expressions see a similar, although not so substantial, simplification to Delete expressions. The `updateLocator` (3.5) non-terminal is used for the lookup, and `'json'` is removed, leaving the replace production rule to only account for the expression containing the value that will replace the old value.

Syntax 3.6: RumbleDB Replace Expressions

Replace Expressions in RumbleDB still use several terminal symbols to make the statement read more like English and so take the form:

```
replaceExpr :  'replace' 'value' 'of' updateLocator
                                        'with' replacer_expr=exprSingle  (3.7)
```

Using the syntax of 3.7, ANTLR4 can easily represent the core components of a replace expression in the AST; in particular, the item to update and the new item to replace the old.

3.1.6 Rename Syntax

As will be seen throughout our implementation, Rename and Replace expressions share much the same shape and logic. Consequently, the Rename and Replace expression syntax are updated similarly, taking full advantage of the `updateLocator` (3.5) syntax.

Syntax 3.7: RumbleDB Rename Expressions

Rename Expressions in RumbleDB are made up of a target item identifying a key to rename, and an item representing the new name of the key, like so:

```
renameExpr :  'rename' updateLocator 'as' replacer_expr=exprSingle
```
(3.8)

3.1.7 Insert Syntax

Insert expressions remain almost identical to their specification counterparts (2.4 and 2.5). However, the `'json'` terminal is removed and ANTLR4 is used to neatly decompose the syntax into the primary elements that comprise the expression, as shown in 3.9 and 3.10

Syntax 3.8: RumbleDB Insert Expressions

RumbleDB describes Insert Expressions as either

```
insertExpr :  'insert' to_insert_expr=exprSingle 'into'
              main_expr=exprSingle ('at' 'position' pos_expr=exprSingle)?
```
(3.9)

or

```
insertExpr :  'insert' pairConstructor (',' pair Constructor)*
                                    'into' main_expr=exprSingle
```
(3.10)

where the `pairConstructor` refers to a key expression and value expression making a key-value pair

3.1.8 Append Syntax

Lastly, the Append Expression adapts its syntax very slightly from the specification (2.8). Syntax 3.11 makes the expression more readable for the query writer and easier to decompose for further development on the AST.

Syntax 3.9: RumbleDB Append Expressions

Append Expressions are designed exclusively for arrays and so further information can be passed through the AST by creating the production rule like so:

```
appendExpr :  'append' to_append_expr=exprSingle 'into' array_expr=exprSingle
```
(3.11)

Once all of the production rules for each new expression are defined they can be added into the `exprSingle` rule, thereby integrating them into the possible syntax that will be identified as an expression. Moreover, having simplified and decomposed the syntax for each expression, the AST created from the grammar will have all the information necessary to easily produce an expression tree.

3.2 Updating Expressions

Expressions in RumbleDB are represented by an abstract `Expression` class that extends an abstract `Node` class acting as a building block of the intermediate representation of a JSONiq query. This `Expression` class describes an arbitrary expression in the expression tree. However, not all nodes in this tree are expressions, as some may be for function declarations and clauses of a `FLWOR` expression. Nevertheless, each expression is associated with a static context containing additional information about the context in which the expression is present, such as in-scope variables and currently known function signatures. Each expression also contains a statically inferred sequence type that describes the sequence that would result from evaluating this expression, but this is only available via static analysis.

3.2.1 Expression Classification

In order to accommodate updating expressions, we assign a new member variable to the `Expression` class called the expression classification, like so:

```
    protected ExpressionClassification expressionClassification =
        ExpressionClassification.UNSET;
```

The `ExpressionClassification` is implemented using a Java Enum, available in appendix A.1, and is RumbleDB's adaptation of the expression classification described for XQuery in subsection 2.3.3. In RumbleDB, expressions share four possible classifications with those of XQuery – simple, basic-updating, updating, and vacuous – as well as an additional *UNSET* classification signifying that the expression has not yet been classified. Moreover, classifications are understood with respect to the JSONiq data model (2.4.2) of Items:

- **Simple expressions:** any expression that is not an updating or basic-updating expression.
- **Basic-updating expressions:** any delete, insert, replace, rename, or append expressions, or a call to an updating function.
- **Updating expressions:** any basic-updating expression, or any expression directly containing an updating expression (recursively defined), except transform expressions
- **Vacuous expressions:** any expression explicitly defined to be vacuous by JSONiq – nevertheless they all either return an empty sequence or raise an error.
- **Unset expressions:** the default classification of all expressions, indicating that they have not been classified and the classification is currently erroneous.

These classifications offer us a finite and static strategy to describe the state of an expression: it is indeed impossible for an expression to change its classification at runtime, since that would require the expression, or its constituents, to morph into a different expression. Hence, an enum was chosen to represent these classifications, as enums are good for encoding a small, finite set of possible states while maintaining readability. Moreover, enums can include methods – such as `boolean isUpdating()` method in appendix A.1 – to encode logic that can be derived from the enum state. A possible alternative is to simply represent the classification with a string, but this would be cumbersome and inextensible. Indeed, checking if an expression is updating would require string parsing, and adding additional classifications would entail an overhaul of all checks of these strings. Another alternative is to use an abstract class to represent an arbitrary classification and employ inheritance to create concrete classifications. However, this approach does not offer us any additional benefits beyond the extensibility of inheritance, which is unnecessary for so few states with little logic since this would add overhead for what could be a simple encoding.

In addition to having an `ExpressionClassification` member variable, we extend the `Expression` class with a `getter` and `setter` interface to retrieve and initialise the expression classification. The `setter` method is used when providing additional information to expressions, such as their classification, during visits to the expression tree. In subsection 2.5.4 we outlined that RumbleDB utilises visitor patterns (Palsberg and Jay, 1998) to traverse the expression tree and apply expression-specific methods. Expression classifications are initialised by one such visitor: the `ExpressionClassificationVisitor` class.

Visitors function by providing a `defaultAction` method for each node in a tree which describes the default logic to apply if no visit method is created for the node currently being visited. Often the default logic is simply to visit all of the descendants of the current expression, i.e. all expressions in the tree that are below the current expression. In the `ExpressionClassificationVisitor`, the default action is also to visit all of the descendants of the current expression and then use their classifications to derive the classification of the current one according to the aforementioned rules. However, very few expressions can be classified as updating without being explicitly specified as being able to be updating by the XQuery Update Facility 1.0, described in subsection 2.3.3. It is therefore simpler to implement specific visit methods for the expressions that can be updating and throw an error (`XUST0001` or `InvalidUpdatingExpressionPositionErrorCode`) if an expression is classified as updating in the default action. The result of this is the `defaultAction` method for the `ExpressionClassificationVisitor` presented in appendix A.2. While the following expressions have specific visit methods implemented to match the semantics demanded by the XQuery Update Facility 1.0 (2.3.3), JSONiq Updates specification (2.4.4), and the RumbleDB classifications: `CommaExpression`, `FlworExpression`, `ConditionalExpression`, `TypeSwitchExpression`, `DeleteExpression`, `InsertExpression`, `RenameExpression`, `ReplaceExpression`, `AppendExpression`, `TransformExpression`, `FunctionCallExpression`, `InlineFunctionExpression`, where the last two expressions fit the criteria due to updating function declarations.

Expression classifications then allow us to extend the `Expression` class with a `boolean isUpdating()` method. The notion of being *updating* is simplified as it is owned by and interfaced via the `Expression` as opposed to the `ExpressionClassification` state.

3.2.2 Expression Classes & Visitors

Each of the syntaxes in section 3.1 will create nodes in an AST that need to be transformed into an expression tree. To do so, we use a `TranslationVisitor` class, similar in structure to the `ExpressionClassificationVisitor` class described in subsection 3.2.1. The goal of the `TranslationVisitor` is to translate each node of the AST into its corresponding `Expression` implementation, and so each of the newly introduced syntaxes requires its own visit method.

Once the expression tree is created, several other visitors traverse it and populate the additional information required by the new expression classes. The `VariableDependenciesVisitor` resolves dependencies between variables and function declarations to ensure that the expressions that make up the dependencies are evaluated in the correct order. For example, if a variable `$x` depends on a variable `$y`, then `$y` must be evaluated before `$x`. In order to resolve these dependencies, a directed acyclic graph (DAG) represents the dependencies and a topological sort over the DAG is used to reorder the declaration into a valid order. The `StaticContextVisitor` populates the static contexts of expressions via a multi-pass algorithm that enables function hoisting. The `InferTypeVisitor` infers and populates the static sequence type of an expression by descending to expressions it does know the type of, like a `StringLiteralExpression` whose type is defined as a singleton sequence containing only a string item. These types then propagate back up the tree during a visit. Finally, the `ExecutionModeVisitor` sets the highest execution mode of the expression to one of `LOCAL`, `RDD`, or

`DATAFRAME`. These execution modes are described in subsection 2.5.5 and are implemented using an `ExecutionMode` enum similar to that of the new `ExpressionClassification` in its use.

3.2.3 Transform Expression Class

A transform expression can be understood to have three main components: a list of copied variables – the `copyDeclarations`; an expression describing the modification to be made to the `copyDeclarations` – the `modifyExpression`; and an expression to be returned from the transform expression, the `returnExpression`. Each copy declaration can be further broken down into a `Name`, with which to refer to the variable, and an expression represented by the `Name`, this being the `sourceExpression`.

We extend the **TranslationVisitor** to decompose the syntax of 3.2 and visit the sub-expressions to create a `TransformExpression` instance with member variables for the `copyDeclarations`, `modifyExpression`, and `returnExpression`. From here, visiting a `TransformExpression` becomes far more involved as this expression introduces variables. To resolve these variables and ensure they are evaluated in the correct order, a `visitTransformExpression` method is implemented in the **VariableDependenciesVisitor**. This transform visit adds all of the dependencies of the new variables, contained in the `copyDeclarations`, to the DAG, ensuring that any declarations are given a working reordering. Without this visit, no `copyDeclaration` could reliably refer to variables declared outside of the transform expression, as the `sourceExpression` of the `copyDeclaration` may try to evaluate a variable reference that has not been properly evaluated yet itself. After `CopyDeclaration` dependencies issues are resolved, the variables can be added to the static context via a specific visit to the `TransformExpression` in the **StaticContextVisitor**. Now the static type of both the `copyDeclarations` and the `TransformExpression` can be inferred by the **InferTypeVisitor**. Naturally, the type of each `copyDeclaration` follows from the associated `sourceExpression`, and the type of the `TransformExpression` is exactly that of its `returnExpression`, and the visit method reflects this logic. Lastly, the **ExecutionModeVisitor** also must provide a specific visit method for a `TransformExpression` as each `copyDeclaration` has an associated mode under which it is stored and this exactly reflects the highest execution mode of the `sourceExpression`. Once more, this is crucial as any access to the `copyDeclaration` must adjust its own execution mode to match that of the `copyDeclaration`.

3.2.4 Inline Function Expression Class

Function declarations are not represented in the expression tree as they are not expressions, so the mechanism for expressing if a function declaration is updating is different from the other expressions that we will discuss. During the translation from the AST to the expression tree, the **TranslationVisitor** uses the function declaration syntax (3.3) to create a `FunctionIdentifier` and an `InlineFunctionExpression`. The `InlineFunctionExpression` is the representation of the declared function in the expression tree and so contains whether the function is updating using the `isUpdating` method of the `Expression` class. While, the `FunctionIdentifier` is, as the name implies, a way to identify the function so it can be inserted into the static context by the **StaticContextVisitor** for future reference, much like variables. In the static context, functions are mapped from their `FunctionIdentifier` to their `FunctionSignature` which is also extended to include whether the function is updating. Subsequently, any call to the declared function – represented by a `FunctionCallExpression` – can look up the `FunctionSignature` to know if the call is updating.

3.2.5 Update Locator Processing

Despite not being an expression itself, the update locator is a crucial part of Delete, Replace, and Rename expressions. The syntax 3.5 shows the two main components of the update locator, those being the primary expression (`PrimaryExpr`) and the one or more array or object lookups. However, this syntax is misleading as the array or object that this update targets is not necessarily the `PrimaryExpr`, but rather the evaluation of all the array and object lookups (except the last) on the `PrimaryExpr`. The last lookup indicates the key/index that will be deleted, replaced, or renamed in the previously evaluated object/array. See figure 3.1 for a pictorial decomposition.

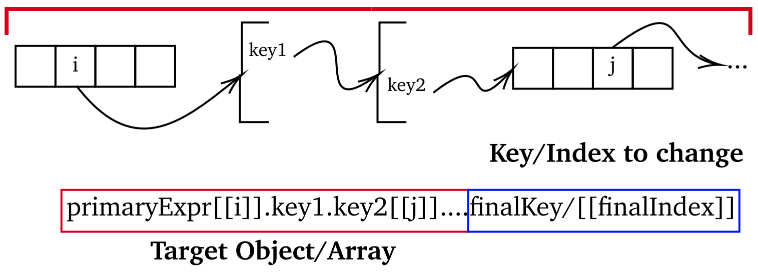


Figure 3.1: Visual breakdown of Update Locator semantics with a depiction of the navigation to the target

To resolve all of these lookups, the **TranslationVisitor** visits the `PrimaryExpr` and then nests it and the following lookups in either an `ArrayLookupExpression` or `ObjectLookupExpression` until the final and most outer lookup is reached. In the expression tree, we dub the first half of the update locator as the *main expression*, which we can think of as a straight line of nodes coming from the initial `PrimaryExpr`. We dub the second half as the *locator expression* and is much simpler as it will always be a single `ArrayLookupExpression` or `ObjectLookupExpression`. In order for expressions containing an update locator to be easily translated, we can extract the main expression and the locator expression using our `getMainExpressionFromUpdateLocator()` and `getLocatorExpressionFromUpdateLocator()` methods introduced in the **TranslationVisitor**.

3.2.6 Delete Expression Class

Illustrated by its simple syntax (Syntax 3.6) the delete expression is processed identically to the update locator. The `DeleteExpression` class looks as if an update locator were formalised into an expression; a `DeleteExpression` instance is comprised of a `mainExpression` and a `locatorExpression` derived from the `getMainExpressionFromUpdateLocator()` and `getLocatorExpressionFromUpdateLocator()` methods in the **TranslationVisitor**. Finally, much like all basic updating expressions, the **InferTypeVisitor** is able to easily infer that any `DeleteExpression` will return an empty sequence as they generate PULs instead.

3.2.7 Replace Expression Class

Replace expressions are very similar to delete expressions as much of their syntax, and so the information that makes up the `ReplaceExpression` class, comes directly from the main expression and locator expression extracted from the update locator processing. Moreover, as a `ReplaceExpression` has the `BASIC_UPDATING` classification, the **InferTypeVisitor** sets its sequence type to empty. The novelty of each `ReplaceExpression` is the inclusion of a `replacerExpression`, being the expression evaluated to replace the value described by the update locator information.

3.2.8 Rename Expression Class

As we have seen with `ReplaceExpressions`, creating the class and instances thereof of a `RenameExpression` follows the same processing as for `DeleteExpressions` (3.2.6) with respect to the **TranslationVisitor** and **InferTypeVisitor**. Nevertheless, a novelty is present in the class as each `RenameExpression` also requires an expression evaluating to the new name of the renamed key – the `nameExpression`.

3.2.9 Insert Expression Class

An `InsertExpression` is implemented with a `mainExpression`, the expression evaluating to the target array or object, a `toInsertExpression`, the expression resulting in the new object or new item to be inserted into the target, and a `positionExpression`, the expression that gives the index to insert the item at in the array. In the first syntax, 3.9, each constituent of an `InsertExpression` instance is clearly defined and obtained by the **TranslationVisitor**, but in the case of an object insertion the `positionExpression` will not be present and so can be null. To account for this possibility, the `InsertExpression` class includes a way to safely check whether the `positionExpression` is null – the `boolean hasPositionExpression()` method – which also acts as a method of discerning between an object insertion and array insertion expression.

Although the second syntax (Syntax 3.10) asserts that the `positionExpression` will always be null, it raises the issue of translating a list of `pairConstructors` into an object to be used as the `toInsertExpression`. To do so, the **TranslationVisitor** considers the left-hand side of each `pairConstructor` as a key and the right-hand side as a value. A list of key expressions and a list of value expressions are generated and combined in an `ObjectConstructorExpression` to form the object to be inserted.

Finally, as with all basic updating expressions, the **InferTypeVisitor** initialises the sequence type of every `InsertExpression` as the empty sequence.

3.2.10 Append Expression Class

The final expression class to create is the `AppendExpression` which can be semantically perceived as a subset of an `InsertExpression` since it is like an array insertion where the index is always the length of the array. An expression that evaluates to a target array, the `arrayExpression`, and an expression resulting in an item to be added to the end of said array, the `toAppendExpression`, are all that make up an `AppendExpression`. Thus, like the `DeleteExpression`, the **TranslationVisitor** can easily decompose and translate the append syntax (3.11) to create an `AppendExpression`, which is then described with an empty sequence type by the **InferTypeVisitor**.

With all of the new expression classes implemented, visitors extended, and a reliable way to describe whether an expression is updating – or simple, etc. – an expression tree representing an updating JSONiq query can be created.

3.3 Updating Iterators

The responsibility for processing and materialising the results of a query falls on RumbleDB's iterators, actualised by the `RuntimeIterator` class. Before this thesis, `RuntimeIterators` would only need to provide an interface for materialising sequences as this is all the JSONiq processing model (2.4.3) would require. However, in order to accommodate updating expressions and the semantics demanded by JSONiq updates (2.4.4), the `RuntimeIterators` must be extended beyond the current implementation described in subsection 2.5.5. Our primary focus of this extension is to include the ability to process PULs and to create `RuntimeIterator` implementations for each of the newly introduced expressions so that an expression tree for an updating query can be transformed into an iterator tree that can process said query.

3.3.1 Extensions to Runtime Iterators

The `RuntimeIterator` class needs two methods to facilitate PUL processing: one must indicate whether this iterator returns a non-empty PUL instead of a non-empty sequence and the other must provide a way to materialise such a PUL – the following methods and member variable accomplish exactly that:

```
protected boolean isUpdating;
.
.
.
public boolean isUpdating() {
    return this.isUpdating;
}

public PendingUpdateList getPendingUpdateList(DynamicContext context) {
    throw new OurBadException(
            "Pending Update Lists are not implemented for the iterator "
                + getClass().getCanonicalName(),
            getMetadata()
    );
}
```

The `isUpdating` member variable is intended to reflect the `ExpressionClassification` (3.2.1) of the expression from which the iterator is derived. For example, iterators derived from expressions with an `ExpressionClassification` of `BASIC_UPDATING` or `UPDATING` will have an `isUpdating` value of `true` to indicate that the iterator is updating and so produces a non-empty PUL. Whereas, iterators derived from expressions with an `ExpressionClassification` of `SIMPLE` will have an `isUpdating` value of `false` to indicate the iterator produces a non-empty sequence. The `isUpdating()` method is then simply a getter for the `isUpdating` state.

The final addition to the `RuntimeIterator` class is the `getPendingUpdateList(DynamicContext context)` method which generates the PUL for the current iterator, but if the iterator is not designed to be able to produce a PUL then an error is thrown to indicate the incorrect use of the iterator. Moreover, the general use of this method is to first check if the iterator `isUpdating()` and if so start processing the PUL, but if not then just return an empty PUL; hence adhering to the extensions to the data model provided by the XQuery Update Facility 1.0 (2.3.3).

Lastly, the `DynamicContext` argument of the `getPendingUpdateList` method is the equivalent of the `StaticContext`, briefly discussed in section 3.2, for a `RuntimeIterator`. Instead of remaining static, the `DynamicContext` updates during the processing of iterators to account for new variables, functions, and additional information introduced by other iterators, thereby enabling a dynamic scope to run throughout the program alongside the `StaticContext` specific to each iterator.

3.3.2 Updating Iterator Classes

Creating the expression tree from the AST involves using a visitor to translate the AST nodes to instances of the `Expression` class. Translating the expression tree to a tree of runtime iterators is no different, and it is the role of the **RuntimeIteratorVisitor** to visit each `Expression` and create a `RuntimeIterator` instance. As a result of the numerous visitors supplying each expression with as much information as required, the methods to translate an `Expression` into a `RuntimeIterator` are very simple. Each of the classes discussed in section 3.2 has its own visit method in the **RuntimeIteratorVisitor**. This method visits every sub-expression of a given updating `Expression` instance and translates it into a `RuntimeIterator` that becomes a sub-iterator of the new updating `RuntimeIterator` instance. For example, in subsection 3.2.6 each `DeleteExpression` instance is shown to be comprised of a `mainExpression` and a `locatorExpression`. The **RuntimeIteratorVisitor** will visit the `mainExpression` and the `locatorExpression` to create a `mainIterator` and a `locatorIterator` that are used to instantiate a `DeleteExpressionIterator`. Alongside this, the `StaticContext` and the `ExecutionMode` of the `Expression` are directly carried over to the new `RuntimeIterator`. This

procedure is the same for each of the new expressions with the exception of the `TransformExpression` (3.2.3), which converts the list of `CopyDeclarations` to a map from the `Name` of the copied variable to the new `RuntimeIterator` that can materialise the associated value. Finally, the `isUpdating()` method of each `Expression`, from which a `RuntimeIterator` is derived, is used to set the `isUpdating` flag of said `RuntimeIterator`.
Our new expressions now have a corresponding `RuntimeIterator` class, and so the expression tree can successfully be translated into a tree of runtime iterators. The next step is to implement the methods for materialising PULs so this iterator tree can process the query result.

3.4 Sequences & Pending Update Lists

Adding the ability to update JSONiq items to RumbleDB involves a large extension to the data and processing model. Both a new type of output, the *Pending Update List (PUL)*, is added and the behaviour of sequences relative to this new output type is altered. Moreover, every expression that can produce a PUL requires an implementation of the corresponding `RuntimeIterator`'s `getPendingUpdateList()` method.

3.4.1 Update Primitives In RumbleDB

Update primitives are the fundamental building blocks for updating JSONiq items in RumbleDB. Each of the seven update primitives describes a different way to alter the state of either an array or an object. Subsection 2.4.4 describes the shape of an update primitive as if it were a function, but this lacks the extensibility and simplicity of a more abstract approach that would aid the production and processing of PULs. Hence, we require a logical interface that decomposes update primitives into their most foundational components. This thesis looks at a **Target-Selector-Content** decomposition of update primitives.

- **Target**: the array or object that is to undergo some update
- **Selector**: the item that identifies the component of the *Target* that is to be updated; this may be an object key with a string type, or it could be an array index with an integer type
- **Content**: the item(s) that introduce new information to the *Target*, possibly at a specified *Selector*

Utilising this decomposition, the update primitives can take on a more homogeneous shape, but still, some heterogeneity arises since not all update primitives require *Content*. The decompositions of the update primitives from subsection 2.4.4 are outlined in figure 3.2.

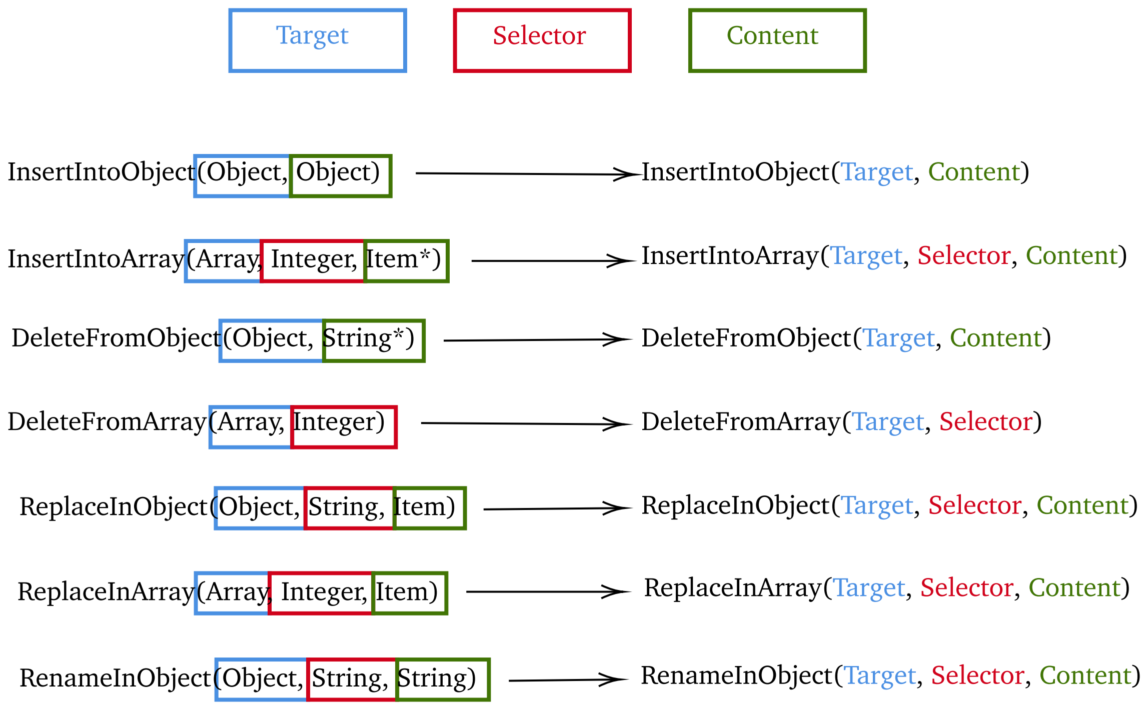

Target
Selector
Content
InsertIntoObject(Object, Object)
InsertIntoObject(Target, Content)
InsertIntoArray(Array, Integer, Item*)
InsertIntoArray(Target, Selector, Content)
DeleteFromObject(Object, String*)
DeleteFromObject(Target, Content)
DeleteFromArray(Array, Integer)
DeleteFromArray(Target, Selector)
ReplaceInObject(Object, String, Item)
ReplaceInObject(Target, Selector, Content)
ReplaceInArray(Array, Integer, Item)
ReplaceInArray(Target, Selector, Content)
RenameInObject(Object, String, String)
RenameInObject(Target, Selector, Content)

Figure 3.2: Target-Selector-Content Decomposition of Update Primitives

Using this model, in conjunction with the `Item` interface (2.5.6), an interface can be created to describe how each update primitive needs to be implemented. Crucially, the `Item` interface can be used as a single type that can abstract away the specific types of any *Target*, *Selector*, or *Content* that may be part of an update primitive. The `UpdatePrimitive` interface mimics the `Item` interface in that it outlines methods to identify the current type of update primitive being represented, for example the `boolean isDeleteObject()` method returns true from the delete-from-object update primitive and false for all others. Moreover, getter methods are provided to request the `Item` representing the *Target*, *Selector*, or *Content*. However, some update primitives have multiple *Contents* and so a special method is available called `getContentList()` to request the list of `Items` representing the *Content*. Lastly, each update primitive must implement a method that applies the changes described by the update primitive, and naturally this is the `void apply()` method. Appendix A.3 outlines a concrete example of an update primitive interface.

Update primitives could have been implemented using a hierarchical inheritance structure with an abstract `UpdatePrimitive` class at the top instead of an interface. However, the only state that every update primitive shares is the *Target*, meaning many of the benefits of an abstract class, like extracting shared logic, cannot be taken advantage of. Moreover, the likelihood is that a query will incur the production of many update primitives, signifying that a class hierarchy would add a lot of processing overhead with little to gain for development purposes.

The `UpdatePrimitive` interface then allows for a class to be created for each update primitive, while ensuring they can all be interfaced with in a similar manner.

The **InsertIntoObjectPrimitive** class contains an `Item target` and an `Item content`, both of which are objects. As the responsibility of update primitives is only to apply an update given arguments that fit their model, no checks for the type of each `Item` are performed and they are assumed to be correct. Thus, when applying the primitive it can be assumed that all methods intended for objects in the `Item` interface are safe to use; the `apply()` method iterates through all of the keys of the `content`, uses `getItemByKey(String key)` to get the value of the key in the `content`, and inserts the key-value pair in the `target` using the `putItemByKey(String key, Item value)` method.

The **InsertIntoArrayPrimitive** class contains an `Item target`, an `Item selector`, and a `List<Item> content`, which are an array, an integer index into the `target`, and a list of items to insert into the `target`, respectively. When instantiating this primitive, the `selector` is checked to be within the bounds of the `target` and, if it is not, an error is thrown to communicate this. To apply this primitive the `Item` interface is extended with a `putItemsAt(List<Item> items, int i)` method that inserts a list of items into an array at a given index. Before calling this method on the `target` with the `content` and `selector`, the `selector` must be decremented by one to account for Java being 0-indexed and JSONiq being 1-indexed.

The **DeleteFromObjectPrimitive** class contains an `Item target`, intended to be an object, and a `List<Item> content`, intended to be a list of the keys to be removed from the `target`. Upon instantiation of this primitive, each key in the `content` is verified for existence against the `target` object using the `getItemByKey(String key)` method of the `Item` interface. If this method returns `null`, then the key is not present in the `target` and an error is thrown to indicate such. In order to apply the primitive, a new method is added to the `Item` interface called `removeItemByKey(String key)` which removes a key-value pair from an object – this method is then applied to the `target` for each key in the `content`.

The **DeleteFromArrayPrimitive** class contains an `Item target`, being an array, and an `Item selector`, being an integer of the index to remove from the `target`. Before the primitive can be created however, the `selector` is checked as to whether it is a valid index into the `target` – if it is not in the range of the `target` then an out of bounds error is thrown. When applied to the `target`, this primitive uses the new `removeItemAt(int i)` method in the `Item` interface to remove the item at the index of the `selector` decremented by one.

The **ReplaceInObjectPrimitive** class contains an `Item target` as an object, an `Item selector` as a string key of the `target`, and an `Item content` as the item to replace the value at the `selector` in the `target`. The `selector` is verified as a key of the `target` by checking if a value is associated with the `selector`, if not then an error is thrown. When applying this primitive, a combination of the `removeItemByKey` method, introduced for `DeleteFromObjectPrimitives`, and the `putItemByKey` method are used to first remove the `selector` from the `target` and then add it back with the new `content`. This avoids unnecessarily creating additional methods that would bloat the `Item` interface.

The **ReplaceInArrayPrimitive** class contains an `Item target` as an array, an `Item selector` as an integer index into the `target`, and an `Item content` as the item to replace the value at the `selector` in the `target`. Once more an out-of-bounds error check is performed on the `selector` with respect to the `target` upon instantiation. Moreover, as was the case with the `ReplaceInObjectPrimitive`, the application of this primitive takes advantage of methods introduced for array deletions and insertions to first remove the item at the decremented `selector` in the `target`, and then insert the `content` at that same index.

Finally, the **RenameInObjectPrimitive** class contains an `Item target` as an object, an `Item selector` as a string key of the `target`, and an `Item content` as the new string key to replace the key at the `selector` in the `target`. The `selector` is error checked as a valid key of the `target`, and applying the primitive is like a hand-over-hand procedure in which the item currently at the `selector` in the `target` is temporarily stored, the key-value pair is removed, and the item is re-inserted into the `target` under the new `content` key.

These update primitive classes are part of the underlying implementation for processing updates in RumbleDB, and so should not be directly interacted with by other components of the system. The `UpdatePrimitive` interface ensures that update primitives can be accessed homogeneously, but for proper encapsulation, the instantiation of each primitive must also be abstracted. Thus, the `UpdatePrimitiveFactory` is introduced, and employs the factory design pattern, described in (Gamma et al., 1995), to instantiate each update primitive. Accessing these methods is done through a `static` instance of the factory, thereby encapsulating update primitives behind the interface and factory so that the processing logic of PULs can be simplified.

3.4.2 Pending Update Lists in RumbleDB

In the context of RumbleDB, PULs, or Pending Update Lists, are unordered collections of update primitives, which represent the change of state intended to be applied to an array or an object with respect to a given snapshot of state. It is through PULs that iterators can create and process updates defined by update primitives. PULs enable this processing via two update routines – `mergeUpdates` and `applyUpdates`, both described in subsection 2.4.4.

To create a **PendingUpdateList** class, the requirements of the primary update routines must be assessed. Both PUL update routines require the categorisation and comparison of the different update primitives in order to function. Importantly, merging update primitives requires that their decomposed forms are merged since many of the merge operations compare the *Targets*, *Selectors*, and *Content* of update primitives within the same snapshot. Whereas applying updates needs the update primitives to be in their composed form such that their `apply()` methods can be executed. Thus, the `PendingUpdateList` class can maintain either a collection of `UpdatePrimitives` or their decomposed versions. For this extension of RumbleDB, the latter was chosen because `applyUpdates` can only be called once per `PendingUpdateList` instance, while `mergeUpdates` can be called many times and the likelihood is that many PULs will be merged before the final merged PUL is applied as many expressions create small PULs that are merged by other expressions, like `FlworExpressions`. Thus, the latter version does not involve the overhead of repeatedly decomposing and recomposing update primitives for each execution of `mergeUpdates`.

In practice, this decomposed variation uses five `Map` instances to match the various shapes of the decomposed update primitives outlined by figure 3.2. Although, figure 3.2 only shows three distinct shapes, five `Map` instances are needed to account for the various *Target*, and *Content* types, which will simplify the `mergeUpdates` and `applyUpdates` algorithms by making this distinction so they need not. Moreover, we do not need as many as seven `Map` instances to account for each type of update primitive since a decomposed `ReplaceInArrayPrimitive` is identical to a decomposed `DeleteFromArrayPrimitive` where the *Content* is null, indicating the removal of the value at the *Selector* in the *Target*. Similarly, a decomposed `DeleteFromObjectPrimitive` can be interpreted as several decomposed `ReplaceInObjectPrimitives` where each of the keys in the *Content* of the `DeleteFromObjectPrimitive` are a *Selector* in a new `ReplaceInObjectPrimitive` with a null *Content*, using the same semantics as with array replacements and deletions. Notably, this same logic cannot also be applied to the `Map` instances needed for insertion primitives since no insertion can be merged with either a deletion or a replacement as defined by the JSONiq updates specification (Robie et al., 2011). Therefore, a strict ordering is required between insertions and other primitives otherwise a single insertion and deletion could share the same *Target* and *Selector* meaning they would need to occupy the same slot if they shared a `Map`. Below is the Java syntax for these `Map` instances in the `PendingUpdateList` class and figure 3.3 shows the equivalence between the update primitives decomposed in their `Map` and their composed version.

```java
    private Map<Item, Item> insertObjMap;
    private Map<Item, Map<Item, List<Item>>> insertArrayMap;
    private Map<Item, Map<Item, Item>> delReplaceObjMap;
    private Map<Item, Map<Item, Item>> delReplaceArrayMap;
    private Map<Item, Map<Item, Item>> renameObjMap;
```

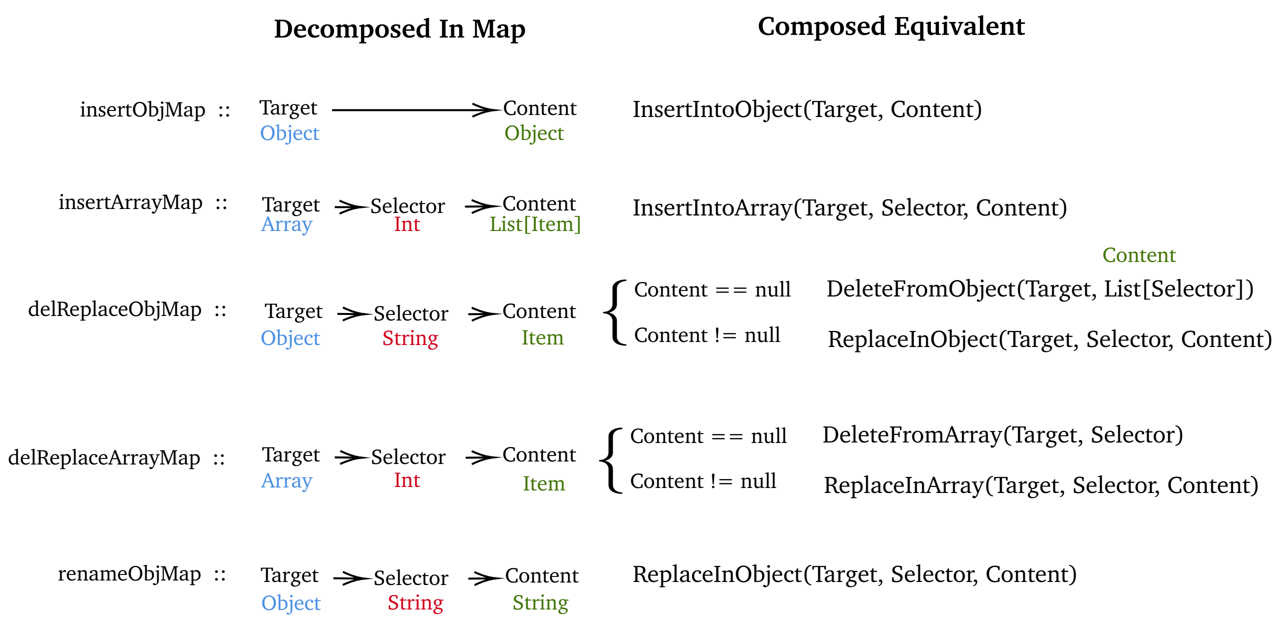

Decomposed In Map
Composed Equivalent
insertObjMap :: Target Object → Content Object
InsertIntoObject(Target, Content)
insertArrayMap :: Target Array → Selector Int → Content List[Item]
InsertIntoArray(Target, Selector, Content)
delReplaceObjMap :: Target Object → Selector String → Content Item
Content == null
DeleteFromObject(Target, List[Selector])
Content
Content != null
ReplaceInObject(Target, Selector, Content)
delReplaceArrayMap :: Target Array → Selector Int → Content Item
Content == null
DeleteFromArray(Target, Selector)
Content != null
ReplaceInArray(Target, Selector, Content)
renameObjMap :: Target Object → Selector String → Content String
ReplaceInObject(Target, Selector, Content)

Figure 3.3: Illustration of Maps for the PendingUpdateList class

These five `Map` instances then make up the state of a `PendingUpdateList` class, and while they could be implemented with either a hash map or a tree map, our extension of RumbleDB opts for a tree map. Tree maps differ from hash maps in that tree maps maintain their entries in a sorted order based on a given comparator using a red-black tree, whereas hash maps use hashing and hash buckets to maintain their entries which provide no particular order. Despite hash maps having a better amortised time complexity for the standard operations – insertions, searches, and deletions all having constant time, while tree maps have $O(\log(n))$ time complexity for the same operations – hash maps use much more memory, and provide no ordering when extracting values, which will prove useful for the update routines of PULs, while minimising the overhead of large PULs. Moreover, hash maps in Java use the `equals()` and `hashCode()` methods of values to enter them into the hash map, and RumbleDB's current implementation already has language-dependent semantics associated with these methods for `Items` meaning they cannot be altered for PULs without disrupting other features. Thus, for two different `Items` `item1` and `item2`, it can be the case that `hashCode(item1) == hashCode(item2)` and that `item1.equals(item2) == true`, meaning that their logical models cannot be relied upon to differentiate them. Instead, the default hash code function can be used, which returns the internal memory address the object was instantiated at, as this is highly unlikely to be the same for two `Items` that are logically equal. However, the default hash code function can only be used in the current implementation as it is designed for local execution. The code that matches this logic is used for the comparator that informs the ordering in the tree maps and can be found in appendix A.4.

Finally, the `PendingUpdateList` class can be instantiated with an `UpdatePrimitive` that is decomposed and entered into the appropriate `Map`. Having prepared the state of a `PendingUpdateList` and created a way to discern between the different decompositions of update primitives, we can now properly implement the update routines.

The **mergeUpdates** routine of a PUL combines the update primitives of two PULs, removes redundancies, and flags errors where appropriate. The goal of this routine is to only have one PUL at the end of every snapshot so no conflicts can arise upon the application of the update primitives. Moreover, thanks to the `Maps` of decomposed update primitives, merging two PULs just involves iterating through the keys of each `Map` of one PUL and processing them relative to the same `Map` of the other PUL, as described by the JSONiq update specification (Robie et al., 2011) and figure 2.5. During the algorithm pseudocode, the PUL being merged into will be referred to as `PUL1` while the PUL merging into `PUL1` will be referred to as `PUL2`. Due to all of the iterations required for this routine, the `Map` accesses are best described in a Java-like pseudocode alongside descriptions of the core components of the merging of each update primitive as adjusted from the descriptions of subsection 2.4.4, utilising the model of update primitives introduced in subsection 3.4.1.

Firstly, the merging of two `delReplaceObjMaps` is concerned with `DeleteFromObjectPrimitives`, `ReplaceInObjectPrimitives`, and `RenameInObjectPrimitives`. Merging two `DeleteFromObjectPrimitives` requires both primitives to have the same *Target* and *Selector*, if so then one can be forgone. On the other hand, two `ReplaceInObjectPrimitives` cannot be merged as no replacement on the same *Target* and *Selector* can take precedence over another, so instead an error is thrown indicating that too many replacements have occurred on the same *Target* and *Selector*. However, a `DeleteFromObjectPrimitive` and a `ReplaceInObjectPrimitive` or `RenameInObjectPrimitive` can be merged as deletions take precedence over replacements and renamings, meaning if the aforementioned pairings of primitives have the same *Target* and *Selector* then only the `DeleteFromObjectPrimitive` remains. Code block 3.1 reflects this logic along with the navigation of the `Maps`.

Code 3.1: Merge Object Deletions & Object Replacements

```
for (target : PUL1.delReplaceObjMap.keySet()) {
    // get inner maps
    pul2SelectorContentMap = PUL2.delReplaceObjMap.get(target);
    pul1SelectorContentMap = PUL1.delReplaceObjMap.get(target);

    for (selector : pul2SelectorContentMap.keySet()) {
        // get content
        pul2Content = pul2SelectorContentMap.get(selector);
        pul1Content = pul1SelectorContentMap.get(selector);
        pul1HasSelector = pul1SelectorContentMap.containsKey(selector
            );
        if (pul2Content == null) {
            // if new content is delete -- remove rename if present
            hasRename = PUL1.renameObjMap.get(target).containsKey(
                selector);
            if (hasRename) {
                PUL1.renameObjMap.get(target).remove(selector);
            }
        } else {
            // if new content is replace check if duplicate or
                redundant
            if (pul1HasSelector && tempSrcRes != null) {
                throw TooManyReplacesOnSameTargetSelectorError();
            } else if (pul1HasSelector) {
                continue;
            }
        }
        // add new content
        pul1SelectorContentMap.put(selector, pul2Content);
    }
    PUL1.delReplaceObjMap.put(target, pul1SelectorContentMap);
}
```

Analogously, `DeleteFromArrayPrimitives` with the same *Target* and *Selector* result in only one remaining when merged, and `ReplaceInArrayPrimitives` with the same *Target* and *Selector* raise an error for the same rationale as the `ReplaceInObjectPrimitive` case. Furthermore, `DeleteFromArrayPrimitives` merged with `ReplaceInArrayPrimitives` result only in the original `DeleteFromArrayPrimitive`. Such similar logic results in a similar algorithm for merging two `delReplaceArrMaps`, as seen in code block 3.2.

Code 3.2: Merge Array Deletions & Array Replacements

```
for (target : PUL2.delReplaceArrayMap.keySet()) {
    // get inner maps
    pul2SelectorContentMap = PUL2.delReplaceArrayMap.get(target);
    pul1SelectorContentMap = PUL1.delReplaceArrayMap.get(target);

    for (selector : pul2SelectorContentMap.keySet()) {
        // get content
        pul2Content = pul2SelectorContentMap.get(selector);
        pul1Content = pul1SelectorContentMap.get(selector);
        pul1HasSelector = pul1SelectorContentMap.containsKey(selector
            );
        if (pul2Content != null && pul1HasSelector) {
            // if new content is replace check if duplicate or
                redundant
            if (pul1Content == null) {
                continue;
            } else {
                throw new TooManyReplacesOnSameTargetSelectorError();
            }
        }
        // add new content
        pul1SelectorContentMap.put(selector, pul2Content);
    }
    PUL1.delReplaceArrayMap.put(target, pul1SelectorContentMap);
}
```

Similarly to `ReplaceInObjectPrimitives`, no two `RenameInObjectPrimitives` can be merged for the same rationale as replacements, meaning this merge would result in an error indicating that too many renames have occurred on the same *Target* and *Selector*. Merging two `renameObjMaps` also needs to account for the removal of renamings that conflict with already present deletions, thus giving the algorithm in code block 3.3.

Code 3.3: Merge Object Renamings

```
for (target : PUL2.renameObjMap.keySet()) {
    // get inner maps
    pul2SelectorContentMap = PUL2.renameObjMap.get(target);
    pul1SelectorContentMap = PUL1.renameObjMap.get(target);

    for (selector : pul2SelectorContentMap.keySet()) {
        // check if duplicate rename
        if (pul1SelectorContentMap.containsKey(selector)) {
            throw TooManyRenamesOnSameTargetSelectorError();
        }
        // check if delete is present
        isDelete = PUL1.delReplaceObjMap.get(target).containsKey(
            selector) && PUL1.delReplaceObjMap.get(target).get(
            selector) == null;
        if (isDelete) {
            continue;
        }
        // add new content
        pul2Content = pul2SelectorContentMap.get(selector)
        pul1SelectorContentMap.put(selector, pul2Content);
    }
    PUL1.renameObjMap.put(target, pul1SelectorContentMap);
}
```

In comparison, two `InsertIntoArrayPrimitives` with the same *Target* and *Selector* merge by concatenating their *Contents* together. The lack of interaction with other update primitives when merging means the merge of two `insertArrayMaps` is simple and independent of other maps, resulting in the pseudocode of code block 3.4.

Code 3.4: Merge Array Insertions

```
    for (target : PUL2.insertArrayMap.keySet()) {
        // get inner maps
        pul2SelectorContentMap = PUL2.insertArrayMap.get(target);
        pul1SelectorContentMap = PUL1.insertArrayMap.get(target);

        for (selector : pul2SelectorContentMap.keySet()) {
            // get content
            pul2Content = pul2SelectorContentMap.get(selector);
            if (pul1SelectorContentMap.containsKey(selector)) {
                pul1Content = pul1SelectorContentMap.get(selector);
                // concatenate content
                pul2Content.addAll(pul1Content);
            }
            // add new content
            pul1SelectorContentMap.put(selector, pul2Content);
        }
        PUL1.insertArrayMap.put(target, pul1SelectorContentMap);
    }
```

Finally, two `InsertIntoObjectPrimitives` may have the same *Target*, and if so merging them only involves inserting the key-value pairs of one *Content* into the other's *Content*. Fortunately, just as seen with `insertArrayMaps`, object insertions are independent of other update primitives. Thus giving a simple algorithm to merge two `insertObjMaps` as shown in code block 3.5.

Code 3.5: Merge Object Insertions

```
    for (target : PUL2.insertObjMap.keySet()) {
        // get content
        pul2Content = PUL2.insertObjMap.get(target);
        if (PUL1.insertObjMap.containsKey(target)) {
            pul1Content = PUL1.insertObjMap.get(target);
            // concatenate content
            pul2Content.insertAll(pul1Content);
        }
        // add new content
        PUL1.insertObjMap.put(target, pul2Content);
    }
```

As the `mergeUpdates` routine is likely to be the update routine most frequently executed, the focus of the `PendingUpdateList` class so far has been on balancing the speed, memory efficiency, and correctness associated with merging PULs. The next challenge is to reconstruct the update primitives and correctly apply them.

The implementation of **applyUpdates** requires an understanding of the difficulties of implementing a declarative language like JSONiq in an imperative language like Java. The definition of PULs provided by the XQuery Update Facility (Consortium et al., 2011) states that PULs are unordered collections of update primitives, and so the order of the application of update primitives does not matter. However, in practice, this is not the case, specifically for array updates. For example, take `A` to be an arbitrary array which has an insertion at index `x` applied to it, and a deletion at index `x + 1` applied in the same snapshot, then the resulting `A` depends on the order of application of these update primitives as shown in figure 3.4.

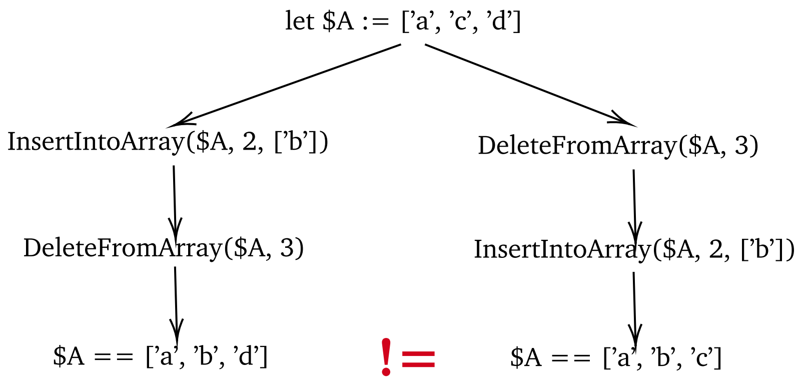


Figure 3.4: Inconsistency of results from the application of unordered update primitives

Related inconsistencies arise from the composition of other array update primitives. Thus, array update primitives are not associative, even with primitives of the same category, meaning some form of ordering must be enforced when maintaining PULs or when applying the updates of PULs. From the example in figure 3.4, it is difficult to ascertain which ordering is correct, but first considering the effect an update primitive has on the indexes of the array will help reveal an appropriate ordering. A `DeleteFromArrayPrimitive` at index `x` will decrement all indexes greater than `x`; an `InsertIntoArrayPrimitive` at index `x` will increment all indexes greater than or equal to `x` so that the inserted value has index `x`; a `ReplaceInArrayPrimitive` makes no changes to the indexes as it only swaps values. Since insertions affect the index `x` while deletions do not, an insertion before deletion will disrupt the intended effect of the deletions, whereas a deletion before insertion will not affect the insertion. Figure 3.5 helps visualise this dependency. Therefore, when considering array update primitives on the same *Selector*, deletions must occur before insertions.

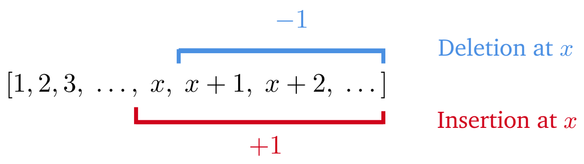


Figure 3.5: Effects of insertions and deletions on array indexes

Observing this same index relationship between update primitives of different *Selectors*, it is clear to see that updates primitives with a `Selector` at index `x` will disrupt update primitives with a `Selector` at an index greater than `x`. Therefore, array update primitives must be applied in descending order of *Selector*, with insertions occurring last the *Selectors* are the same. With this rubric, of the orders in our initial example in figure 3.4, the right-hand path should be chosen since the *Selector* of the `DeleteFromArray` is greater than that of the `InsertIntoArray`.

For update primitives on objects, however, more nuances arise. Objects have no natural ordering to their keys, meaning that there is no natural way to order the update primitives. As a result of this, each update primitive on an object has to be validated against the original `Target` independently of other update primitives, meaning that their order does not matter as long as they are independently validated. For example, take an arbitrary object `O`. If a deletion at key `x` is applied in conjunction with an insertion of key `x`, it is either the case that `O` already contains the key `x` or it does not. If `O` does not contain `x` then the deletion is invalid, but if `O` does contain `x` then the insertion is invalid – in either case, an error is thrown and so no order is valid. This same logic can be applied to replacements and renamings, but when merging update primitives deletions supersede both so the example only considers deletions and insertions. In order to keep some consistency between applying update primitives to arrays and objects, the chosen order for applying update primitives to objects matches that of arrays with renamings appearing last. Hence, object update primitives will be applied in the order `DeleteFromObjectPrimitive`/`ReplaceInObjectPrimitive`, followed by `InsertIntoObjectPrimitive`, then finally `RenameInObjectPrimitive`.

The last consideration is the reformation of the `UpdatePrimitive` classes from their decomposed form in the `Maps` of the PUL. To do so, each `Map` is iterated over and the `UpdatePrimitive-Factory` is used to instantiate the corresponding `UpdatePrimitive`. If this `UpdatePrimitive` is to be applied to objects then it can be added to a list of other object update primitives. Otherwise for arrays, a tree map from *Target* to a map linking each *Selector* to a list of `UpdatePrimitves` is used to maintain the descending order of *Selectors* over the same *Target* array. Moreover, the inner list of `UpdatePrimitives` is added to first by iterating over the `delReplaceArrayMap` and then over the `insertArrayMap` to ensure that even within the same *Selector*, insertions come last. Each of these iterations over a `Map` is almost identical to the corresponding pseudocode in code blocks 3.1 to 3.5, with the exception that the innermost logic is not a merge operation but rather the aforementioned population of lists and maps of `UpdatePrimitive` instances. Once each list has been compiled, it can be iterated over using the `apply()` method of each `UpdatePrimitive` to enact the update.

Now that we have both the `mergeUpdates` and the `applyUpdates` routines implemented, we can finally use the `PendingUpdateList` to process and store `UpdatePrimitives`.

3.4.3 Mutability Scoping

Before implementing the materialisation of PULs, it must first be understood which items can be updated and which cannot. According to the XQuery Update Facility (Consortium et al., 2011) and JSONiq update specification (Robie et al., 2011), only those items that have been copied by a transform expression (3.2) can be updated. Moreover, while the variables introduced in the `copy` clause of a transform expression can be referenced throughout the expression they can only be modified by the `modify` clause of the same expression. This means that copied variables cannot be modified by nested transform expressions. Thus, the scope in which copied variables can be changed is extremely limited.

To account for this scoping, the notion of a *Mutability Level* is introduced to RumbleDB, wherein each transform expression incurs a new mutability level. Both items and scopes have mutability levels, represented by an `int` piece of state named `mutabilityLevel`, and items can only be updated when their `mutabilityLevel` matches that of the scope they are in. As seen, items are implemented with the `Item` interface, and these scopes are tracked through both `StaticContext` and `DynamicContext` instances. Moreover, the `mutabilityLevel` of scopes increases linearly with the scoping of transform expressions, where a `mutabilityLevel` of `0` indicates that the program is not inside of a transform expression and a `mutabilityLevel` of `n` indicates that the program is in `n` nested transform expressions. From here, instances of `Item` have a `mutabilityLevel` of `-1` when not associated with the `copy` clause of a transform expression, otherwise the `mutabilityLevel` is set equal to that of the transform expression's scope. This logic is illustrated by figure 3.6.

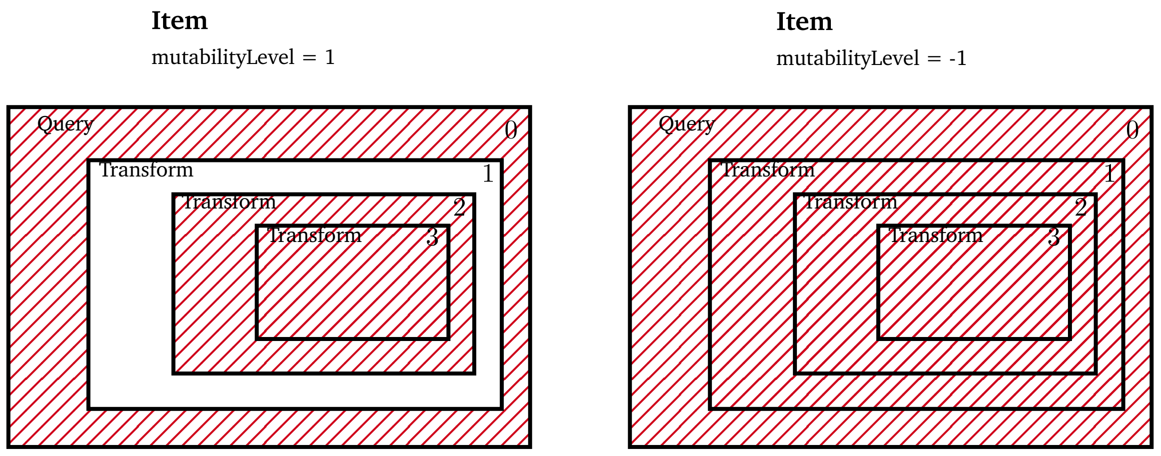


Figure 3.6: Example of Mutability Level restricting scopes in which an item can be updated (red indicates immutable)

Using this `mutabilityLevel`, we can dynamically check the validity of updates as the iterator tree materialises the PULs. However, to make this `mutabilityLevel` available, several aspects of RumbleDB must be extended, namely the `Item` interface, the `StaticContext` class, the `DynamicContext` class, and of course both the `TransformExpression` and `TransformExpressionIterator` classes.

The first step is to give the `StaticContext` class a `currentMutabilityLevel` along with getters and setters so that when the `StaticContextVisitor` visits a `TransformExpression` the `currentMutabilityLevel` of the `StaticContext` can be incremented at the start and decremented at the end of the visit. Thus maintaining the logic of updating the mutability level in accordance with the scopes of transform expressions. Then, the `TransformExpression` must similarly be extended with a `mutabilityLevel` that can be set during the visit of the `StaticContextVisitor` using the `currentMutabilityLevel` of the `StaticContext`. Now the expression tree has the information about the mutability levels of every `TransformExpression` it contains. Once the state of the `TransformExpressionIterator` class has been extended to include a `mutabilityLevel`, this information can be directly passed onto the `TransformExpressionIterator` during the translation from the expression tree to the iterator tree described in section 3.3.

Finally, to allow the `TransformExpressionIterator` to interface with `DynamicContext` and `Item` instances during PUL materialisation, both are extended with getters and setters for a mutability level. Similar to `StaticContexts`, each `DynamicContext` maintains an always updating `currentMutabilityLevel` as part of its state. Whereas not all items need a mutability level, as only those that can be updated – objects and arrays – can have non-negative mutability levels. Therefore, only the state of the `ObjectItem` and `ArrayItem` classes include a `mutabilityLevel`.

3.4.4 Pending Update List Materialisation

The materialisation of PULs is similar to that of sequences, described through execution modes in subsection 2.5.6. The `RuntimeIterator` class is equipped with a method indicating that PUL materialisation is possible, `isUpdating`, and a method to materialise a PUL, `getPendingUpdateList`. It is the second method that completes the brunt of the processing and must be implemented for each updating iterator, i.e. the `RuntimeIterator` representing each `Expression` that could have the `BASIC_UPDATING` or `UPDATING` `ExpressionClassification` (3.2.1), as well as `TransformExpressions`. For each of the `UPDATING` expressions, that is to exclude all of the new expressions, the `getPendingUpdateList` method is similar to their `getRDDAux` method and follows the rough structure of code block 3.6 in which `getPendingUpdateList` is invoked on each child iterator and the resulting PULs are all merged.

Code 3.6: Updating Expression PUL Materialisation

Materialisation of a PUL from an updating iterator that was not derived from a `BASIC_UPDATING` expression, in which an empty PUL is immediately returned if the iterator is not updating:

```
if (!iterator.isUpdating()) {
    return new PendingUpdateList();
}
PendingUpdateList pul = new PendingUpdateList();
for (RuntimeIterator child : iterator.children) {
    pul.mergeUpdates(child.getPendingUpdateList(context));
}
return pul;
```

Nevertheless, the implementations still have nuances to offer, especially with respect to the new expressions.

Although a **TransformExpressionIterator** does not produce a PUL upon its return, it does need to create a PUL from the `modify` clause that must be applied so that the changes to the copied variables are available in the `return` clause. To materialise this PUL, the `DynamicContext` must be prepared before being used by the `getPendingUpdateList` method of the iterator representing the `modify` clause (the `modifyIterator`). To do so, the value associated with each copy variable is materialised into a sequence (a list of items) that is cloned using Apache Commons (Apache Commons Team). Each cloned `Item` then has its `multabilityLevel` set to the `mutabilityLevel` of the current `TransformExpressionIterator`, so that it may be modified. Once the cloned sequence is prepared, it can be associated with the name of the copy variable in the `DynamicContext`. Then the `currentMutabilityLevel` of the `DynamicContext` can also be updated to that of the current `TransformExpressionIterator`, and so this context can be used as the context from which the PUL of the `modify` clause is generated. The resulting PUL of the `modifyIterator` is the resulting PUL of the `TransformExpressionIterator`.

The **StaticUserDefinedFunctionCallIterator** is the iterator derived from the `InlineFunctionExpression` (3.2.4) which is the expression derived the `FunctionDeclaration` syntax (3.1.2). The implementation of `getPendingUpdateList` for this iterator is very similar to the generic implementation in code block 3.6, with the exception that the iterator representing the body of the function being called must be looked up in the `DynamicContext` before `getPendingUpdateList` can be invoked. As only one iterator is required here there is no need to merge any PULs.

The **ReturnClauseSparkIterator** class holds the final steps when processing a FLWOR expression in RumbleDB. FLWOR expressions can be thought of as a river that tuples float down in order to reach the `ReturnClauseSparkIterator`, and along the way, they may become beached when they do not meet the requirements of a where-clause, or they may clump together due to a group-by-clause – in any case, processing a FLWOR expression means processing the tuples that reach the return-clause. The local execution implementation of `getPendingUpdateList` is no different. For each tuple, a new

`DynamicContext` is prepared which adds all of the necessary variable bindings ready for the expression in the return-clause to generate a PUL for that tuple. As every tuple generates a PUL, they must be merged, and the final PUL resulting from the merge is then the output PUL for an updating `ReturnClauseSparkIterator`. Once more this procedure is very similar to code block 3.6, but with different tuples instead of different child iterators.

The **TypeswitchRuntimeIterator** class must first discern the correct branch that would be selected by the type of the expression being switched upon. Once the branch is known the corresponding iterator can be used to materialise the resulting PUL for the `TypeswitchRuntimeIterator`. Here, instead of all child iterators being taken into account as in code block 3.6, only one child iterator needs to be processed meaning no merging of PULs is required.

The **IfRuntimeIterator** class has an implementation of `getPendingUpdateList` very similar to the `TypeswitchRuntimeIterator` but must only discern between two iterators, associated with either the `then` clause or the `else` clause. When the correct clause is known, `getPendingUpdateList` is invoked on the associated iterator to materialise the resulting PUL.

The **CommaExpressionIterator** class has an implementation of `getPendingUpdateList` nearly identical to code block 3.6, as all that is required is to merge the PULs resulting from the constituent expressions.

The following iterators correspond to `BASIC_UPDATING` expressions and so share commonalities. These are the terminal iterators for materialising PULs; they create the `UpdatePrimitive` instances that are then added to a `PendingUpdateList`. The PULs produced by their `getPendingUpdateList` method will only contain one `UpdatePrimitive` as, currently, each of these expressions can only describe an update to one item at a time. Moreover, it is in these iterators that the *Target*, *Selector*, and *Content* are materialised and their types are verified. The validity of the mutability of the *Target* is also verified. The general flow of these error checks is: if the *Target* is an object, then the *Selector* is verified as a string and the `mutabilityLevel` of the *Target* is checked to be equal to that of the current `DynamicContext`; if the *Target* is an array, then the *Selector* is verified as an integer and the `mutabilityLevel` is checked to be equal to that of the current `DynamicContext`. Any other circumstances will result in an error describing the issue, and these errors correspond to those mentioned throughout subsections 2.3.3 and 2.4.4. This general flow of error checking is outlined in code block 3.7.

Code 3.7: Basic Updating Expression PUL Materialisation

```
Item target = targetIterator.materialize(context);
Item selector = selectorIterator.materialize(context);
Item target = contentIterator.materialize(context);

PendingUpdateList pul = new PendingUpdateList();
UpdatePrimitive up;

if (main.isObject()) {
    if (!selector.isString()) {
        throw CannotCastUpdateSelectorException();
    }
    if (target.getMutabilityLevel() != context.
        getCurrentMutabilityLevel()) {
        throw TransformModifiesNonCopiedValueException();
    }
    up = factory.createObjectUpdatePrimitive(target, selector,
        content);
} else if (target.isArray()) {
    if (!selector.isInt()) {
        throw CannotCastUpdateSelectorException();
    }
    if (target.getMutabilityLevel() != context.
        getCurrentMutabilityLevel()) {
        throw TransformModifiesNonCopiedValueException();
    }
    up = factory.createArrayUpdatePrimitive(target, selector, content
        );
} else {
    throw InvalidUpdateTargetException();
}
pul.addUpdatePrimitive(up);
return pul;
```

DeleteExpressionIterators follow the flow of code block 3.7 with the exception that the `UpdatePrimitive` instantiated by the factory is either a `DeleteFromObjectExpression` or `DeleteFromArrayExpression`. However, deletion primitives each only take two arguments, and their decomposed update primitive models do not align, meaning that no explicit *Selector* or *Content* item is materialised. Instead, the item materialised as an integer becomes the *Selector* for a `DeleteFromArrayPrimitive`, and the item materialised as a string becomes the *Content* for a `DeleteFromObjectPrimitive`.

ReplaceExpressionIterators also follow the flow of code block 3.7 and create PULs with either a `ReplaceInObjectExpression` or `ReplaceInArrayExpression`. However, they still require some nuance since any replacement introduces an `Item` to either an object or an array. As a result of this the *Content* must be cloned, like copy variables in a transform expression, to avoid multiple unintentional references to the same underlying `Item` being present in the program.

RenameExpressionIterators can only produce PULs with a `RenameInObjectPrimitive`. These iterators closely align with code block 3.7, but since they can only create `RenameInObjectPrimitives`, the flow associated with the *Target* being an array is not required.

InsertExpressionIterators materialise PULs with either an `InsertIntoObjectPrimitive` or an `InsertIntoArrayPrimitive`. This materialisation requires two main distinctions from code block 3.7. Firstly, only the `InsertIntoArrayPrimitive` requires a *Selector*, meaning that this is only materialised if the associated iterator exists. Secondly, both of these primitives will add an `Item` into either an object or an array and so, for the same reasoning as with `ReplaceExpressionIterators`, the *Content* must be cloned.

AppendExpressionIterators do not have any `UpdatePrimitive` of their own namesake as they instead produce PULs with an `InsertIntoArrayPrimitive`. Thus, they follow the same procedure as `InsertExpressionIterators` by cloning their *Content*, but they only require the array-related flow of code block 3.7 and the *Selector* is known implicitly as the size of the *Target* array incremented by one. Thereby ensuring the insertion occurs at the end of the array to match the logic of appending.

Once each possible updating expression is equipped with a runtime iterator that can materialise PULs, updates can be processed by RumbleDB. The final task is to ensure that sequences safely interact with these PULs and updating expressions.

3.4.5 Sequences in an Updating World

Sequences in RumbleDB are implemented with the `SequenceOfItems` class described in subsection 2.5.6, and were previously the only possible result from evaluating expressions. With the introduction of PULs, expressions produce both sequences and PULs, but crucially, both cannot be non-empty.

Moreover, when a non-empty sequence is output, nothing needs to explicitly happen with the items inside beyond displaying them, unless the query writer specifies otherwise – such as being output to a given file format. Whereas, when a non-empty PUL is output, all of the update primitives within must be applied to end the snapshot associated with the PUL. The same thing happens for PULs generated by the `modify` clause of transform expressions before the sequence of the `return` clause is materialised. Thus once an iterator tree is created from a JSONiq query, if the root iterator is updating, as indicated by the `isUpdating` method, then a PUL should be materialised and applied. However, all of this happens in the `Rumble` class which is used to generate a `SequenceOfItems` and interface with the results of a query. As such it is still possible that this `SequenceOfItems` could be used to materialise a sequence (`List<Item>`) with undefined results. To remedy this issue, maintain the processing model described by (Robie et al., 2011), and avoid overhauling the `Rumble` class which could have many dependencies, the semantics of the `SequenceOfItems` are adjusted. This adjustment simply ensures that if the root iterator used to materialise the sequence is updating, then any attempt to materialise a sequence will only return an empty sequence. RumbleDB's implementation of sequences can then match the semantics of (Robie et al., 2011) and safely coexist with pending update lists.

Thus concludes the implementation of the primary aim of this thesis. Now let us consider the secondary aim.

3.5 Delta Table Integration

Subsection 3.4.5 indicates that when non-empty PULs are output from the iterator tree, they are immediately applied, but because a non-empty sequence cannot also be output, the changes as a result of the PUL cannot be seen without the use of a transform expression. Two methods for remedying this pitfall are available: either scripting is introduced to JSONiq and RumbleDB – for example, an updating script can be followed by a non-updating script that uses the changes of the former script – or the updates can be persisted beyond RumbleDB. As the hope is for RumbleDB to be well integrated into the Big Data ecosystem while ensuring its power can be used beyond its bounds, the latter remedy is chosen.

Persisting updates beyond RumbleDB requires some transactional system to actually store the data in a way that allows for changes to be applied. As discussed in section 2.6, Delta Lake (Armbrust et al., 2020) is the best choice for RumbleDB due to its ACID transactions and easy integration with the Spark API. Integrating the API and capabilities of Delta Lakes will not require an overhaul of the system built for updates so far. Rather, RumbleDB must be extended to handle the files of a Delta Lake table, identify which `Item` instances come from a Delta table, and adapt each `UpdatePrimitive` to be able to be applied to Delta table `Item` instances.

3.5.1 Delta Files

RumbleDB is equipped with several functions that allow query writers to process data from several different file formats from JSON to Parquet. Moreover, RumbleDB can output the results of a sequence – as long as it was executed using the Spark API – to many file formats. In order for RumbleDB to integrate Delta tables, it must first be able to take Delta tables as inputs and be able to output them.

Delta tables use Parquet files to store the data and a write-ahead log to track transactions on the data. This means that Delta tables have schemas and so, to interact with them using the Spark API, the execution must be done via dataframes. Using Delta tables as inputs requires creating a function that takes a file path in the form of a string as input and outputs a sequence where each item is a different row in the Delta table. This function will take the form `delta-file(file_path)`. Adding this function to RumbleDB does not involve any new syntax as it just uses the syntax of a standard function call; instead, the catalogue of built-in functions is extended with the name `"delta-file"` associated with a runtime iterator prepared to process this function – the new `DeltaFileFunctionIterator` class. With this addition to the built-in function catalogue, when the `RuntimeIteratorVisitor` encounters a `FunctionCallExpression` with an identifier matching the name `"delta-file"`, the `DeltaFile-FunctionIterator` can be immediately found as the correct iterator to be added to the iterator tree. When instantiated, the `DeltaFileFunctionIterator` takes a `RuntimeIterator` instance for each of its arguments, of which there is only one that will materialise the file path of the Delta table.

Since Delta tables must be executed in the `DATAFRAME` execution mode, the `DeltaFileFunc-tionIterator` need only implement a `getDataFrame` method to fulfil its promises of execution. In this method, the Delta table file path is materialised by the only argument iterator and used to read the Delta table into a Spark dataframe, which is then wrapped in a `JSoundDataFrame` instance to form the result.

On the other side of interacting with Delta tables, outputting them proves to be very simple with the Spark API. As long as the sequence produced from a JSONiq query can be formulated using dataframes, a Spark `DataFrameWriter` can be used to write the dataframe result to a designated file path. Outputting to a Delta table only requires that a query creates results with a homogeneous schema that can be used by a dataframe and that the query is run with the command line arguments `"--output-format delta"` and `"--output-path $path"`.

3.5.2 Item Identifiers

During the local execution of PULs, each `Item` instance was identified by a combination of its logical `hashCode` value and its physical `hashCode` value. However, when incorporating Delta tables, the execution is no longer restricted to being local, meaning a solely logical method to uniquely identify `Item` instances is needed.

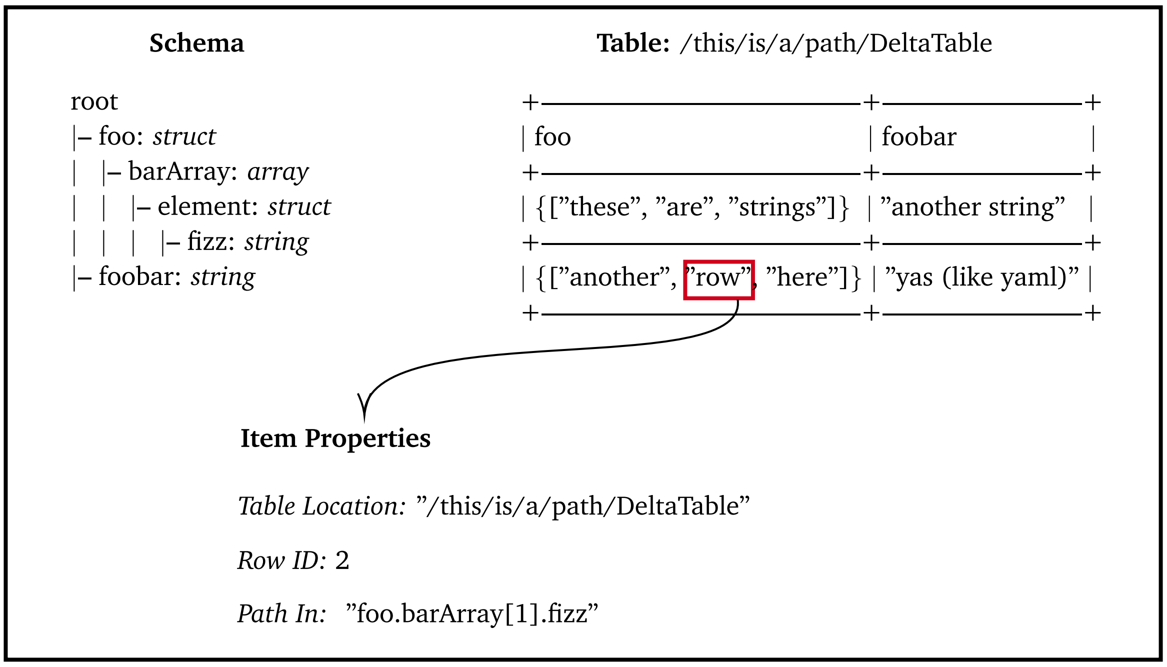
To identify an `Item`, the translation from a Delta table to a sequence must first be understood. A Delta table is made up of rows and columns, where each column has a data type corresponding to a SparkSQL data type as defined by the schema of the table. Every data type available in a Delta table must then be mapped to the equivalent JSONiq data type which is depicted by table 3.1.

Delta Table Data Type	Spark Data Type	JSONiq Item Type
INT	IntegerType	IntItem
BIGINT	LongType	IntegerItem
FLOAT	FloatType	FloatItem
DOUBLE	DoubleType	DoubleItem
STRING	StringType	StringItem
BOOLEAN	BooleanType	BooleanItem
DATE	DateType	DateItem
TIMESTAMP	TimestampType	DateTimeStampItem
ARRAY	ArrayType	ArrayItem
STRUCT	StructType	ObjectItem

Table 3.1: Type equivalences between Delta tables, Spark, and JSONiq

Using table 3.1 each row in a Delta table can then be mapped to an `ObjectItem` where the keys of the object are the names of the table columns and the values are the corresponding row values once they have been mapped to an `Item`. Naturally, this can create very nested objects given a Delta table schema containing many nested columns of `STRUCT` and `ARRAY` types.

Every `Item` derived from a Delta table can be identified by three values. The first and simplest is the actual Delta table that produced the `Item`, which is represented by a string containing a path to the Delta table – this is known as the *Table Location*. The second is the number of the row from which the `Item` came in the Delta table – this is the *Row ID* as an integer. Finally, the third is which column, no matter how nested, the `Item` was mapped from – this is the *Path In* as a string of the series of object and array accesses required to arrive at the column. An example of how these properties correspond to a value in a Delta table is illustrated by figure 3.7.



Schema

```
root
|– foo: struct
|   |– barArray: array
|   |   |– element: struct
|   |   |   |– fizz: string
|– foobar: string
```

Table: /this/is/a/path/DeltaTable

```
+----------------------------------+--------------------+
| foo                              | foobar             |
+----------------------------------+--------------------+
| {["these", "are", "strings"]}    | "another string"   |
+----------------------------------+--------------------+
| {["another", "row", "here"]}     | "yas (like yaml)"  |
+----------------------------------+--------------------+
```

Item Properties

Table Location: "/this/is/a/path/DeltaTable"

Row ID: 2

Path In: "foo.barArray[1].fizz"

Figure 3.7: Example of item properties derived from a Delta table and schema in Spark dataframe notation

Integrating this method of identifying each `Item` begins by extending the `Item` interface to have getters and setters for the three new pieces of state: `tableLocation`, `rowID`, and `pathIn` – with default values of `"null"`, `-1`, and `"null"` respectively. Only the `ObjectItem` and `ArrayItem` need to implement this state since they are the only possible targets of an update primitive. The next challenge is to extract these values when materialising an `Item` from a dataframe. To do so the dataframe must first have these values to extract, thus when a dataframe is created from a Delta file in the `DeltaFileFunctionIterator`, three new columns are added: a `tableLocation`, a `rowID`, and a `pathIn` column. However, the `pathIn` value is not guaranteed to be static because sequences represented by dataframes can be navigated via three means: object lookups (e.g. `seq.key`), array lookups (e.g. `seq[[index]]`), and array unboxings (these "explode" or transform arrays into sequences of the same items and look like `seq_of_arr[]`). Each of these sequence navigation techniques will access a different column in the dataframe, meaning the value of `pathIn` must be updated to match.

Object lookups are processed using the `ObjectLookupIterator`, which, when dealing with dataframes, uses a SparkSQL query to select the desired key from the dataframe, producing a new dataframe. When encountering dataframes from Delta tables, this method is extended to also select the `rowID`, the `pathIn` concatenated with `".$key"`, and the `tableLocation`. With this concatenation to the `pathIn` column, the object lookup is tracked.

Array lookups use the `ArrayLookupIterator` to navigate dataframes by performing a SparkSQL query similar to that of object lookups, with the exception that these lookups select for the array column indexed at a given index (e.g. `SELECT $array[$index] ...`). This too is extended to select the newly added columns for Delta tables, but `pathIn` is instead concatenated with the array indexing, i.e. `"[$index]"`.

Lastly, array unboxings are implemented with the `ArrayUnboxingIterator`. This iterator unboxes an array by performing a select with the SparkSQL `explode()` function applied to the array, which creates a new row in the resulting dataframe for each element in the array. Once more this query is altered to include the columns for Delta table dataframes, but concatenating `pathIn` is not so simple as no static index can be used. Thus, for Delta table dataframes, the SparkSQL `posexplode()` function is used to provide both the position in the array and the array element on a new row. From here the position can be concatenated to `pathIn` to reformulate the array indexing that would have happened otherwise.

The final consideration is that there is no guarantee that the Delta table will have a column for the `rowID` already built in to be matched against the `rowID` of an `Item`. To account for this possibility, before reading the Delta table into a dataframe in the `DeltaFileFunctionIterator`, a `rowID` column must also be inserted into the Delta table.

Materialising an `ObjectItem` or `ArrayItem` from a row of a Delta table dataframe still involves mapping all columns, except the newly added ones, to an `Item` to form the values of the object or the elements of the array. However, the `tableLocation`, `rowID`, and `pathIn` columns must not be mapped to an `Item`. Rather the values associated with these columns should be set as the corresponding state for the resulting `ObjectItem` or `ArrayItem`. With this model, each `Item` instance can be directly associated with a particular value in a Delta table.

3.5.3 Updating Delta Tables

Adapting the `UpdatePrimitive` classes to be able to be applied to `Item` instances from a Delta table is the final step to persisting JSONiq updates. Before implementing the class extensions needed, the notion of the *Mutability Level* must be altered to support updating these Delta `Item` instances. Fortuitously, a gap in the *Mutability Level* model is already present wherein no `Item` can have a `mutabilityLevel` of `0` to match the scope outside of any transform expression. Thus, `Item` instances derived from a Delta table will have a `mutabilityLevel` of `0` such that they can be updated, but not within any transform expression. This too then gives meaning to the topmost expression returning a PUL that is immediately applied, as this application would persist to a Delta table. Integrating these new *Mutability Level* semantics only involves adding a column to the Delta table dataframe called `mutabilityLevel` that has a static value of `0`. The new `mutabilityLevel` column can be added and mapped onto an `Item` in exactly the same way the `tableLocation` column was, for example, in subsection 3.5.2.

The original `apply()` method for the `UpdatePrimitive` interface assumes that the update is performed locally and on `Item` instances as the *Target*, but the real *Target* when persisting to a Delta table is a particular column and row as represented by an `Item`. Therefore, a delineation must be made between applying an `UpdatePrimitive` to an `Item` and applying an `UpdatePrimitive` to a Delta table. Two new methods are introduced to the `Item` interface, `applyItem()` and `applyDelta()`, with the original `apply()` method choosing between the two based on whether the *Target* `Item` has a non-default value for its `tableLocation`. Each concrete `UpdatePrimitive` must now implement the `applyDelta()` method. These methods use the `.sql()` function of the Spark API which executes a provided query with respect to the specified tables. Thus, each update primitive can have a template for a SparkSQL query associated with it, where the blanks are filled in by the *Target*, *Selector*, and *Content* as necessary. However, several difficulties with this approach arise. Firstly, unlike the keys of a `STRUCT` type column, the individual elements of `ARRAY` type columns in a Delta table cannot directly be updated, instead the whole `ARRAY` must be updated at once. Secondly, some update primitives will also update the schema of the Delta table. Thus, each update primitive may be associated with up to four different queries: a query to update the schema of a non-`ARRAY` type column; a query to update the schema of an `ARRAY` type column; a query to change the value of a non-`ARRAY` type column; and a query to change the value of an `ARRAY` type column.

Before we can identify which update primitives will alter the schema, let us understand the meaning of each update primitive as applied to a Delta table. Firstly, a `DeleteFromObjectPrimitive` interprets the keys to delete as columns to set to `NULL` for its *Target* row, because deleting the key for one `Target` row should not delete the key for every `Target` row. Next, an `InsertIntoArrayPrimitive` can only insert *Content* that matches the type of the *Target* array since the Delta `ARRAY` type does not support heterogeneous types. Likewise, a `ReplaceInArrayPrimitive` can only replace an element with *Content* that coincides with the type of the *Target* array. The `InsertIntoObjectPrimitive` can insert *Content* of any supported type but it must first update the schema by adding columns for each new key and then setting the value of all rows except the *Target* row to `NULL`. Whereas, the `ReplaceInObjectPrimitive` can only replace a value with *Content* whose type matches that of the column being replaced on the *Target* row. However, the `RenameInObjectPrimitive` is comparable to the `InsertIntoObjectPrimitive` in that it too must update the schema by introducing a column for the new name *Content* and set the value of all non-*Target* rows to `NULL`, but the type of this new column must match that of the old column name in order to support the old values. Finally, the `DeleteFromArrayPrimitive` remains semantically the same as it need not care for the type of the element it removes from a `Target` array. From these semantics, we recognise that only the `InsertIntoObjectPrimitive` and the `RenameInObjectPrimitive` can alter the schema of a Delta table.

Knowing which update primitives can alter schema highlights our control flow for these `applyDelta()` methods and shows us that most update primitives only require two queries, one for updating `ARRAY` type columns and one for non-`ARRAY` type columns. Updating an `ARRAY` type column occurs when the `pathIn` for the *Target* `Item` includes any array indexing because Delta tables cannot update the individual elements of an array. Therefore, even if the final *Selector* is not an array index we may still need to update a column nested inside of an `ARRAY` type column. Every update primitive then needs a query updating an `ARRAY` type column and a non-`ARRAY` type column. Fortuitously, we can distinguish between the two by checking if the `pathIn` involves array indexing.

Now that we have established the different forms of SparkSQL queries we can consider their general forms with respect to the decomposed model of update primitives from subsection 3.4.1. Code block 3.8 outlines the template for a SparkSQL query that updates a column, regardless of if it is of an `ARRAY` type or not.

Code 3.8: General SparkSQL Column Update

Template for updating a column in a Delta table using SparkSQL and the decomposed update primitive model, where + indicates string concatenation:

```
UPDATE $target.tableLocation
SET $target.pathIn + $selector = $content
WHERE rowID == $target.rowID
```

Depending on the update primitive, the *Target*, *Selector*, and *Content* are arguments for the query, however in our implementation, they are stored as `Item` instances which cannot be interpreted by SparkSQL. Hence, we extend the `Item` interface with a `getSparkSQLValue(ItemType itemType)` method to translate `Item` instances into a string of their corresponding SparkSQL values. This method is implemented for each concrete `Item` type and recurses through components of structured items, such as the keys and values of an `ObjectItem` and the elements of an `ArrayItem`. Most concrete `Items` can just be converted to their string representations, but the key-value pairs of an `ObjectItem` must be wrapped in the SparkSQL `named_struct()` function, while the elements of an `ArrayItem` use the `array()` function. The `itemType` argument of the `getSparkSQLValue` method is used to account for values that must appear in the struct corresponding to an `ObjectItem` but do not appear in the `ObjectItem` because they are `NULL` in the Delta table. Now with this `getSparkSQLValue` method and the knowledge that when an array indexing occurs we need to update the whole array at once, we note that every update involving an array index can be interpreted as mapping that array from the Delta table to an `ArrayItem` containing our *Target*, invoking `applyItem()` on our *Target*, and using `getSparkSQLValue` on the `ArrayItem` to reconstitute the Delta array. Using this method, every application of an update to an `ARRAY` type column can use the same method, thereby simplifying the control flow.

In comparison, queries that alter the table schema do not directly interact with either the *Content* or *Target* `rowID` of the update primitive. Instead, they need only know the location of the table, the location of the new column to add, the name of the new column, and the type associated with that column, as depicted by code block 3.9.

Code 3.9: General SparkSQL Schema Update

Template for updating the schema a Delta table using SparkSQL and the decomposed update primitive model, where + indicates string concatenation:

```
ALTER TABLE $target.tableLocation
ADD COLUMNS ($target.pathIn + $selector $type)
```

Both code block 3.8 and 3.9 show that the *Target*, *Selector*, and *Content* still make up the foundations of our update primitives. However, when updating the schema, the *Content* is present in the form of a `type` argument as opposed to the SparkSQL representation of the underlying `Item`. This `type` must be the SparkSQL representation of the type of our underlying *Content* `Item`. Hence, we must extend the `Item` interface with a `getSparkSQLType` method, analogous to the `getSparkSQLValue` method. This `getSparkSQLType` method is also implemented for each concrete `Item` and returns a string representation of the Delta type equivalent for each JSONiq item, as outlined by table 3.1. The final nuance of schema updates is that if the `pathIn` of our *Target* does contain array indexing then we cannot directly use the `pathIn` to make our update. However, unlike column updates, schema updates do not need to alter the whole `ARRAY` type at once. Rather, each array index in the `pathIn` can be replaced by a `".element"` string to refer to the homogeneous type of the elements of the array, thereby equating the `pathIn` to a series of only object lookups and short-circuiting the schema update.

Now we may implement the `applyDelta()` methods. Firstly, we look towards updating the value of an `ARRAY`-type column as this is common among all update primitives and can be implemented with a `default arrayIndexingApplyDelta()` method in the `UpdatePrimitive` interface. We begin by splitting the `pathIn` value at the first use of array indexing – the first half of this split will give us the path to the array we will update in the Delta table (`preArrayPathIn`) and the second half will be the path in the array to our *Target* (`postArrayPathIn`). Then we extract the array from the Delta table using a `SELECT` with the `preArrayPathIn` and map the resulting dataframe to an `ArrayItem` and its `ItemType`. Next, we navigate the `ArrayItem` using the `postArrayPathIn` until we reach either the object key or the array index that identifies the Delta table clone of our *Target* `Item`. This is not the real *Target* `Item` instance as it was mapped from a dataframe. From here we can invoke `applyItem()` to update the real *Target* `Item` and replace the clone with the real instance. The `ArrayItem` will now contain our updated *Target* `Item` and `getSparkSQLValue` can be applied to the `ArrayItem` to create the *Content* argument of the query template in code block 3.8. Additionally, the `$target.pathIn + $selector` will be replaced with the `preArrayPathIn` to match the array we are updating.

The `InsertIntoObjectPrimitive` must precede the `arrayIndexingApplyDelta` method with a method that first updates the schema when array indexing is involved – the `arrayIndexingUpdateSchemaDelta` method. We start this method by replacing all array indexings in `pathIn` with `".element"` justified by the trick mentioned before. Moreover, we change our template from code block 3.9 by deriving the `$selector` from the key of our *Content* `ObjectItem` and the `$type` by invoking `getSparkSQLType` on the corresponding values of our *Content* `ObjectItem`. Thus we form one query for each key to insert and execute them one by one, skipping any columns that already exist. Correspondingly, the `RenameInObjectPrimtive` also implements an `arrayIndexingUpdateSchemaDelta` method. Once more we replace the array indexings of `pathIn` with `".element"`. However, here our `$selector` argument is the *Content* of the `RenameInObjectPrimtive` as this is the new name of the column, and the semantics of renames in a Delta table dictates that we must add a new column with the new name. Moreover, the `$type` argument is extracted from the Delta table using an SQL `DESC` statement on the original column (the unaltered `pathIn` + the *Selector* of the `RenameInObjectPrimitive`). The resulting query is only executed if the column does not already exist.

Looking towards queries that do not involve an `ARRAY` type column, we must implement an `applyDelta` method for each update primitive. Generally, the control flow of an `applyDelta` method differentiates between `ARRAY` type execution and non-`ARRAY` execution by checking the `pathIn` for array indexing like so:

```
    if (pathIn.contains("[")) {
        if (requiresSchemaUpdate()) {
            arrayIndexingUpdateSchemaDelta();
        }
        arrayIndexingApplyDelta();
    } else {
        ...
    }
```

Each update primitive has a different process for the `else` branch above. Fortuitously, we can refer back to 3.8 frequently in the coming explanations as the `$target` argument and its identifiers are exactly as described in 3.8, since no array indexing is present in the `else` branch. Many of the update primitives for objects share similarities and so we will focus on them first. The `DeleteFromObjectPrimitive` mimics the code block of 3.8 but performs several `SET`s where each `$selector` is a key to be deleted, as described by the *Content*, and each `$content` is `NULL` to indicate that the column has been deleted for this row. Comparably, the `ReplaceInObjectPrimitive` is almost identical to code block 3.8 as the `$selector` and `$content` arguments match with the primitive's *Selector* as a string and the *Content* `Item` reconstituted by the `getSparkSQLValue` method respectively. On the other hand, the `RenameInObjectPrimitive` still needs to update the schema. As with the `ARRAY` type column case, we do so by deriving the type of the new column with the SQL `DESC` statement applied to the original column and input this as the `$type` argument of the template in code block 3.9, with the `$selector` being the new name of the column. Then we set the value of the new column with the template of code block 3.8, where the `$selector` is the new column name and the `$content` is the SparkSQL value of the old column. Moreover, the `InsertIntoObjectPrimitive` also updates the schema but uses the keys and types of the values of the *Content* `ObjectItem` as the `$selector` and `$type` arguments of code block 3.9 respectively. Then we set each of these fields using the template from code block 3.8 where the `$selector` is, again, the *Content* key and the `$content` is the SparkSQL value of the *Content* value.

Finally, we consider the `else` branch for update primitives that update arrays. The `DeleteFromArrayPrimitive`, `ReplaceInArrayPrimitive`, and `InsertIntoArrayPrimitive` share a simplified version of the process required in the case of the updating of an `ARRAY` type column, as we can exploit their `applyItem` methods. For each, we extract the array from the Delta table with a `SELECT` at the column identified by the `pathIn` value, map this array to an `ArrayItem` and its type, invoke `applyItem` to update our *Target* `ArrayItem`, and formulate the updated SparkSQL array by calling `getSparkSQLValue` on the updated `ArrayItem` with the derived type. This updated SparkSQL array then constitutes the `$content` argument for the template in code block 3.8.

Thus concludes the queries for each `UpdatePrimitive` and we can then execute them using the Spark API `sql()` method.

The culmination of integrating Delta file I/O, identifying `Item` instances in a Delta table, and extending each `UpdatePrimitive` to handle being applied to a Delta table then allows us to persist updates while maintaining the simplicity and versatility of JSONiq in RumbleDB. Moreover, the data independence offered by RumbleDB enables us to hide the complexity of not only persisting these updates but also of handling updates in JSONiq at all. Now we look toward benchmarking and experimentation to discover optimisations that can be made.

4 Experimentation

RumbleDB is intended to be a tool for teaching and processing in a Big Data paradigm. Although the aim of this thesis is to introduce updating functionality to RumbleDB and so sheer performance is not a primary goal, it is still crucial that we analyse the performance of updates in RumbleDB against similar systems so that we may understand possible optimisations. Throughout our experimentation, we will be querying Delta tables and so using the extension to JSONiq updates in RumbleDB introduced in section 3.5. Doing so ensures that we explore how RumbleDB may be used for Big Data updates which is the most likely use case for updates in RumbleDB, and after all, if an update is not persisted, did it even update at all?

4.1 Structure

Experimentation requires a structure to rely on to enable repeatability and reproducibility. In this section, we will define the datasets used to evaluate updates in RumbleDB, the systems RumbleDB is compared against, the queries that stretch and showcase the capabilities of RumbleDB's updates, and the way in which our experiments will be conducted.

4.1.1 Datasets

There are two facets of RumbleDB's updates that are crucial to explore. The first is its ability to handle standard queries in an *On-Line Transaction Processing* (OLTP) paradigm, in which updates are commonplace. Often OLTP will focus on relational data and the associated workflows as this has been how data has been structured for much of data management. Therefore, understanding how JSONiq updates in RumbleDB operate under an OLTP paradigm will provide a baseline for the performance of RumbleDB's updates on very structured and homogeneous as compared to more heterogeneous data. Moreover, an OLTP dataset gives some insights into the capability of updates in RumbleDB to adjust to different data shapes. The second facet of RumbleDB's updates to explore is its capabilities with respect to semi-structured and highly heterogeneous data as this is precisely what RumbleDB and JSONiq are designed to process and describe.

Fulfilling the role of the relational dataset for more OLTP queries is the TPC-C dataset (Council). TPC-C is an OLTP benchmark for medium-complexity transactional loads simulating the workload of a wholesale supplier. The dataset is comprised of nine relational tables: `Warehouse`, `District`, `Stock`, `Customer`, `Order`, `History`, `New-Order`, `Item`, `Order-Line`. Each `Warehouse` serves ten `Districts` which themselves serve 3,000 `Customers` who each have at least one `History` and at least one `Order`. Then each `Order` is associated with at most one `New-Order` and five to fifteen `Order-Lines`. Finally, every `Warehouse` has a `Stock` of 100,000 `Items`, this `Stock` then serves at least three `Order-Lines`. For the purposes of our updates, we only care about the shape of our data and the size of our data. All of the tables above are standard relational tables so they are ideal for investigating the OLTP paradigm. The size of each table we will use can be found in table 4.1. This data was acquired in CSV format from the Relational Dataset Repository described and maintained by (Motl and Schulte, 2015). To convert each table from CSV to a Delta table we can use a validation expression from JSONiq and read the CSV file into RumbleDB, validate each record against the expected type, and output the result to a Delta table. Simple examples of validation are available in (rum), and appendix A.6 shows a more complex example of schema validation along with how to output to a Delta table.

Table	Size (# Tuples)
Warehouse	1
District	10
Customer	30,000
Order	30,000
New-Order	9,000
Stock	100,000
Item	100,000
Order-Line	300,000

Table 4.1: Size, in number of tuples, of each table in the TPC-C dataset

Satisfying our need for a heterogeneous and denormalised dataset is the *GitHub Archive* (GA) dataset (gha). This dataset records information about GitHub events, such as the type of event, the author of the event, the associated URL and much more. The exact schema of each record is highly heterogeneous, enough such that the data cannot be put into a dataframe due to a lack of rigidity, meaning that we cannot test the data using our Delta table updates. However, this heterogeneity only persists between different event types, so we will only use the data from push events. In order to acquire this data, the GA website suggests using Google's BigQuery, described by (Fernandes and Bernardino, 2015), which is Google's cloud website for querying Big Data in an SQL-like fashion. BigQuery allows up to 1GB of data to be downloaded as a JSON file in which the field causing the heterogeneity, that BigQuery cannot natively handle, is stored as a string. However, this does not pose a problem for us as there is still plenty of heterogeneity present in the other fields to perform our experiments. For our purposes, this JSON data must be transferred to a Delta table and can be done so via a JSONiq query ran on RumbleDB that reads the JSON data and outputs a Delta table, an example of which is found in appendix A.6. The exact schema for this Delta table is available in appendix A.5. Finally, from this Delta table, we create several copies, each with a power of two tuples to show the relationship between the amount of data and the query execution time. In order to keep execution time realistic while still showing the relationship, the powers of two chosen are 2, 4, 8, 16, 32, 64, and 128.

4.1.2 Systems & Queries

Now that we know which datasets we will be using and why we are examining them, we must assess how we will query them and with what we will compare RumbleDB.

For the TPC-C dataset, we will examine RumbleDB against SparkSQL via the Java API. SparkSQL acts as a good benchmark for the best possible result for RumbleDB as much of the implementation for updating Delta tables uses the SparkSQL Java API.

The GitHub Archive dataset, however, will only be queried using RumbleDB and SparkSQL as PostgreSQL offers very little native support for updating the values of deeply nested data, which can be a struggle for SparkSQL too.

Our queries for the TPC-C dataset are extracted from the TPC-C benchmark specification (Council). We only choose three of the updating queries as very few of the queries actually alter values and all of them are not nested, making them simple relative to RumbleDB's capabilities. The explanation and associated transaction of each query is as follows:

Q1A **Query 1A**: Within the *New Order Transaction*, increment the `NO_O_ID` field of the `New Order` table. (JSONiq: A.7, SparkSQL: A.8)

Q1B **Query 1B**: Within the *Payment Transaction*, add `343` to the `C_BALANCE` field of the `Customer` table, and if the customer has bad credit (`C_CREDIT` is equal to `'BC'`) then add information about the update to `C_DATA`. (JSONiq: A.9, SparkSQL: A.10)

Q1C **Query 1C**: Within the *Delivery Transaction*, sum the `OL_AMOUNT` field of the `Order Line` table for each district, and add this to the `C_BALANCE` of every customer in each district. (JSONiq: A.11, SparkSQL: A.12)

When querying the GitHub Archive dataset we can be much more exploratory. I was unable to find any reference updating queries for the GA dataset, presumably due to it normally serving as a static archive, so I developed the following five queries to stretch the potential of updating highly heterogeneous data.

Q2A **Query 2A**: Toggle the `public` boolean field of each record. (JSONiq: A.13, SparkSQL: A.14)

Q2B **Query 2B**: Give each record's `repo` a nickname and popularity rating!. (JSONiq: A.15, SparkSQL: A.16)

Q2C **Query 2C**: Change the structure of each `id` by refactoring the `id`, `repo.id`, and `actor.id` into a struct in an array to allow for versioning. Remove the old ids. (JSONiq: A.17, SparkSQL: A.18)

4.1.3 Benchmarking Methodology

The final step to being able to analyse the performance of updates in RumbleDB is to devise a framework for executing our queries. This framework requires four inputs for each benchmark. The first is the updating query to be executed, for JSONiq this must be a JSONiq file, but for SparkSQL the input can either be a string query or a `void` Java function that executes the query. The second is the expected location of the table being updated, as referenced by the query. The third is the query responsible for creating the table to be updated. The fourth, and final, is the location of the file to append the results. These inputs then construct a `FileTuple` class instance which can be used by the benchmark framework to execute each input as required. Once initialised, each benchmark will be executed three times without recording the execution time to warm up any cache or memory that may be used. Then the benchmark is executed and recorded, in milliseconds, ten times with the time for each iteration being output, along with the total, mean, and standard deviation at the end. Notably, with each iteration, outside of any timing, the table being updated is created at the start and deleted at the end to reset any automatically created indexes that the table may make. Moreover, Delta tables log all transactions and implement versioning meaning that with each iteration the Delta table will begin to bloat and introduce an unwanted dependence between the iterations. Thus, constantly creating and deleting the table ensures the independence of each iteration.

Finally, all benchmarks are run on my laptop because we are searching for an understanding of how to optimise the implementation, which is only local, so a large computing cluster or a supercomputer is unnecessary. My laptop has the following relevant specs:

- **CPU**: 12th Generation Intel® Core™ i7-1260P Processor (E-cores up to 3.40 GHz P-cores up to 4.70 GHz)
- **Memory**: 16 GB DDR4-3200MHz (Soldered) & 32 GB DDR4-3200MHz (SODIMM)
- **Storage**: 1 TB SSD M.2 2280 PCIe Gen4 Performance TLC Opal

4.2 Performance Evaluation

The structure provided by section 4.1 enables us to decide what we want to measure and then gather our data for analysis.

4.2.1 Performance Metric

Every query we will run is designed to explore the performance of RumbleDB updates across its diverse set of methods to update a variety of shapes of data. Thus, we look toward the mean execution time over ten iterations for each tuple count for each query as the metric that highlights performance since speed is a major limiting factor when processing Big Data. While measuring the execution time we vary the number of tuples being updated by the query in order to assess how RumbleDB's performance changes as the amount of data being processed increases. Comparing this to handwritten SparkSQL queries will allow us to visualise how much optimisation is possible as RumbleDB uses SparkSQL in its implementation, meaning the mean execution time for SparkSQl represents the lower bound of RumbleDB. Lastly, we display the mean execution time and the tuple count on $\log_2$ scales because we measure the mean execution time while doubling the tuple count. Hence, a $\log_2$ scale allows us to better understand the relationship between the two variables.

4.2.2 TPC-C Evaluation

In this subsection, we only discuss the results of Q1B in figure 4.1 as the results for Q1A and Q1C display an almost identical relationship, but they are available in figures A.1 and A.2 in appendix A.19. We see that as the tuple count grows the mean execution time for SparkSQL remains relatively constant at about one second of execution time. This result can be attributed to SparkSQL's ability to scale and parallelise meaning the increase in tuple count was too small to incur any performance penalties. However, for RumbleDB the mean execution time roughly doubles with each doubling in the tuple count, primarily due to our implementation using volcano-style processing to generate one update primitive per record being updated. As a result of this, we perform a SparkSQL query for each record instead of one SparkSQL query batching similar updates together. Therefore, our RumbleDB results can be interpreted as a function of the mean of SparkSQL's mean execution time and the tuple count. Moreover, it is important to note that SparkSQL's mean execution time for two tuples lies at one second while RumbleDB's lies at four seconds. Supposing that RumbleDB's performance was simply SparkSQL's performance multiplied by the tuple count then we would expect this result for RumbleDB to lie at two seconds. However, this additional overhead can be explained mostly by the need for RumbleDB to add a `rowID` column to any Delta table read in by the `delta-file` built-in function. Updating the Delta table in this way always requires altering the schema, which is more computationally expensive than an update to a column. The relationships we have identified here represent how RumbleDB performs on flat, structured, relational datasets. On this kind of data, RumbleDB's expressivity cannot be realised, as seen by the similarities between the JSONiq and SparkSQL queries.

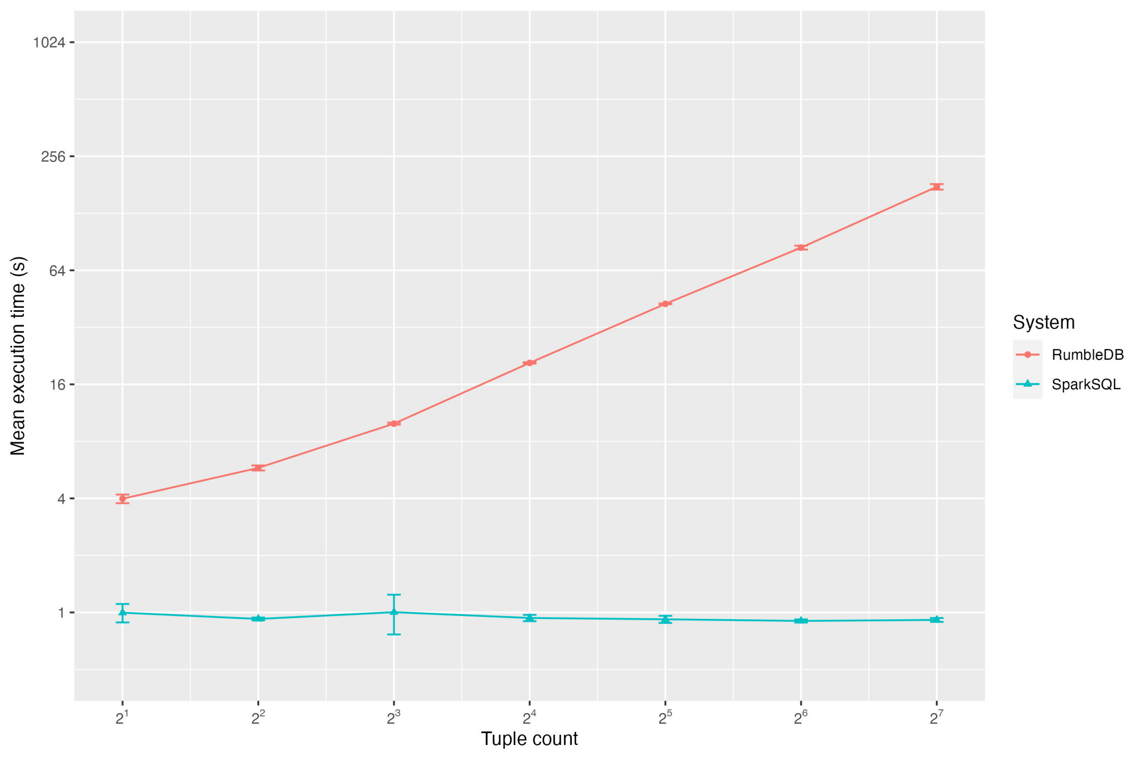


Figure 4.1: Comparison of performance on Query 1B (Q1B) between RumbleDB and SparkSQL, on $\log_2$ scales

4.2.3 GitHub Archive Evaluation

Analogously to our evaluation of performance on the TPC-C dataset in subsection 4.2.2, in this subsection we only display the results for Q2B in figure 4.2 as the results for Q2A and Q2C display the same relationships – their results can be found in figures A.3 and A.4 in appendix A.19.In a similar fashion to the results of 4.1 in figure 4.2 we see SparkSQl remain constant in spite of the increasing tuple count, but here it remains constant at just below two seconds of mean execution time. Likewise, the mean execution time for RumbleDB at a tuple count of two is higher for Q2B than for Q1B and this is likely due to the Q2B adding columns into the schema – a computationally expensive operation – while Q1B only updates an existing column. However, as the tuple count increases for RumbleDB, this additional cost is diminished by the increasing number of column updates because the alteration to the schema only occurs once for each query execution but the column updates occur for each tuple. Once more we see that RumbleDB's mean execution time increases linearly with the tuple count, and this too is due to the one-item-at-a-time updating philosophy of the iterator processing in our implementation. Additionally, the disparity between the mean execution time at a tuple count of two between RumbleDB and SparkSQL is due to the addition of the `rowID` column when using Delta tables. Nevertheless, it is in the code for Q2B in JSONiq and SparkSQL that we see the potency of RumbleDB as SparkSQL requires two separate queries while JSONiq only needs one. This is due to the dataset being naturally heterogeneous and thus more intuitively processed by RumbleDB.

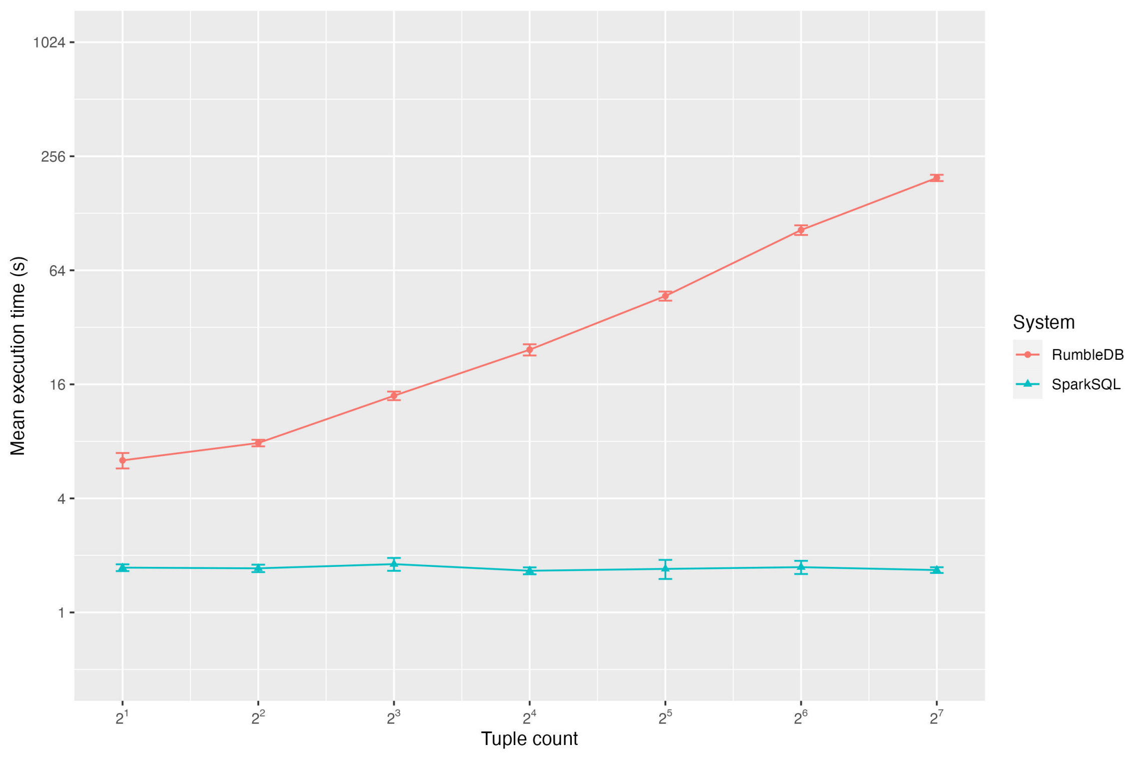


Figure 4.2: Comparison of performance on Query 2B (Q2B) between RumbleDB and SparkSQL, on $\log_2$ scales

4.2.4 RumbleDB Evaluation

Finally, now that we understand the deficiencies of RumbleDB in comparison to SparkSQL, we can investigate how RumbleDB handles the different queries.

Let us begin with figure 4.3 which compares RumbleDB's performance for each query applied to the TPC-C dataset. Q1A and Q1B have very similar mean execution times for each tuple count despite Q1B introducing some control flow in the form of an if expression. This highlights that the processing of PULs, as it flows through the iterator tree, incurs very little computational costs for the repeated application of the `mergeUpdates` routine – implying the algorithm and implementation of PULs is scalable both with more expressions and more tuples, at least for the data measured. The trend for Q1C corroborates this conclusion as we see the cost of computing a join cause the mean execution times for Q1C to begin much higher than either of the other queries. However, as the cost of updating each tuple individually overshadows the cost of the join, we see each query's trend begin to converge. Hence, we can assume that, with careful optimisation, each of these queries can have comparable computational costs with regard to the processing of updates.

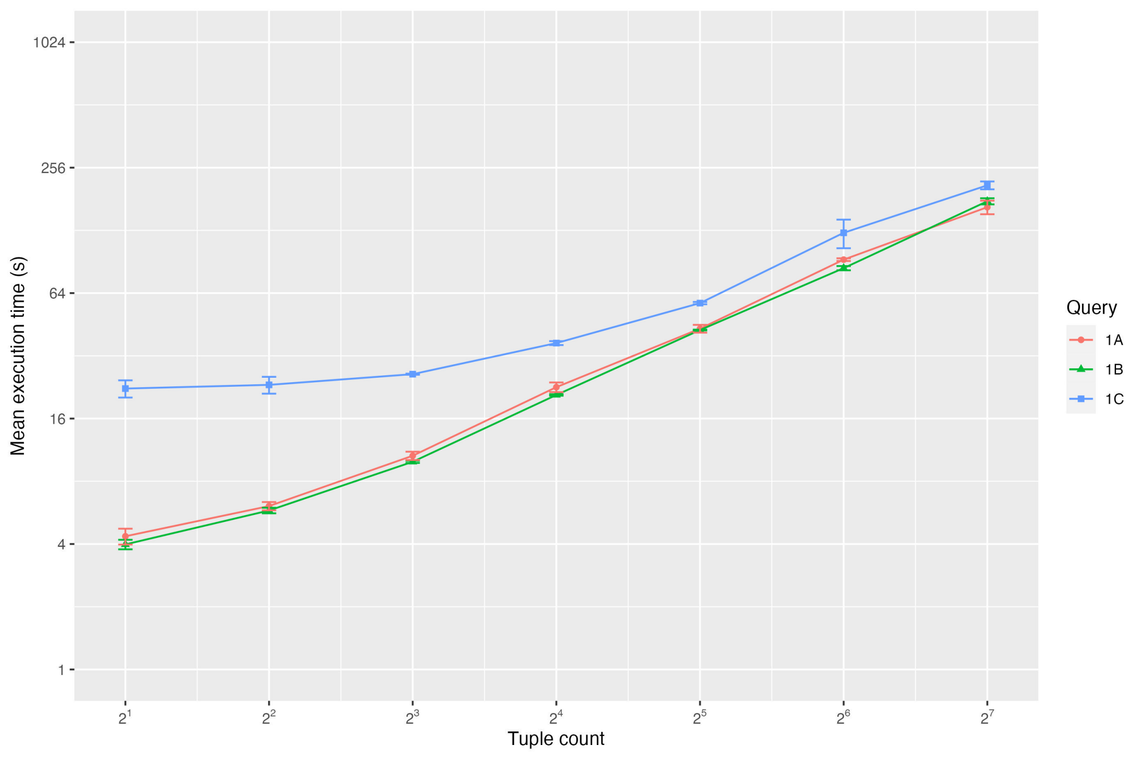


Figure 4.3: Comparison of performance of RumbleDB for Queries 1A, 1B, and 1C (Q1A, Q1B, Q1C), on $\log_2$ scales

Figure 4.4, on the other hand, compares RumbleDB's performance for each query applied to the GitHub archive dataset and outlines trends different from those that occur when updating purely flat datasets. Initially, we see the mean execution time of each query increases linearly with the tuple count as identified in subsection 4.2.3. Nevertheless, on close inspection, we can recognise that Q2A consistently outperforms Q2B, but this superior performance becomes less pronounced as the tuple count increases. This is due to Q2B performing a nested insert, which will alter the schema and update a column, while Q2A only replaces the value in a non-nested column. From this, we realise that no optimisation will be able to equalise the cost of performing these two different types of updates – schema-altering and non-schema-altering – as they are fundamentally disparate. However, this additional cost of Q2B is small and only noticeable due to the small number of tuples being affected, so when processing many more records the difference is likely to be unimportant. However, Q2C has a much larger execution cost associated with it than either of the other two queries do and this is due to it inserting and updating values nested inside of an array. As mentioned in subsection 3.5.3, any update that occurs inside of an array forces the array to be fully materialised, manipulated, and reconstituted as a Delta table array. Consequently, much more computational and memory overhead is required to handle this materialisation, and as the arrays grow or become more nested even more overhead is added. Compounding this is yet again the fact that each record incurs the same processing due to the current model of the update primitives only updating one item at a time. Therefore, it is likely that Q2C would see the largest improvements upon optimisation as they could help batch both the updating of items and the materialisation of large arrays.

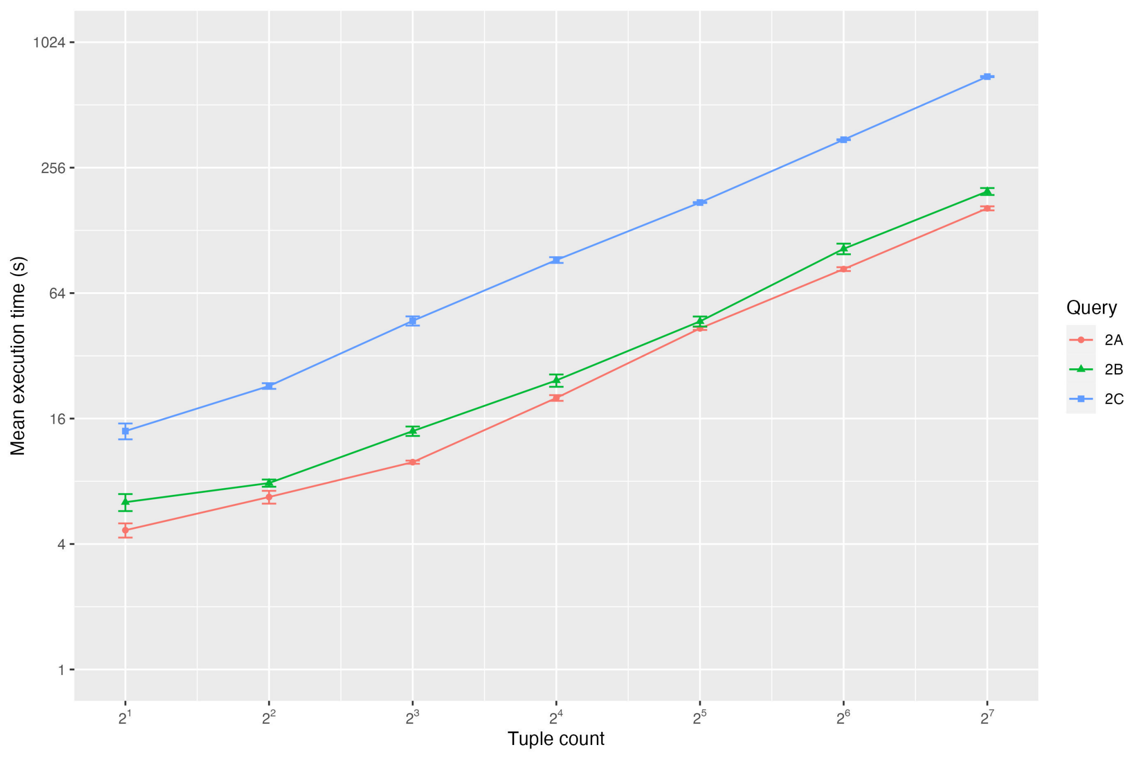


Figure 4.4: Comparison of performance of RumbleDB for Queries 2A, 2B, and 2C (Q2A, Q2B, Q2C), on $\log_2$ scales

Finally, we look towards figure 4.5 to understand how the size of tuples affects RumbleDB's performance. Here we compare RumbleDB with itself for Q1A and Q2A as these are both similar queries in that they only update the top-level columns. The `New Order` table of the TPC-C dataset has 3 columns, each of them 4-byte integers meaning the Delta table will store 12 bytes per tuple. Whereas, the GitHub Archive table has 17 string columns, 3 struct columns, and 1 boolean column, with each tuple averaging 365 bytes. Clearly, in both the number of columns (and so the number of `Item` instances needed to represent a tuple in RumbleDB) and the number of bytes the tuples of the GitHub Archive are larger than the tuples of the `New Order` table. Hence, if we were batch-processing then we would expect similar mean execution times between Q1A and Q2A up until a certain number of tuples at which point memory would start bottlenecking Q2A. However, figure 4.5 shows that the mean execution times of the two queries are virtually the same for each tuple count indicating that the current iterator-processing model in which we only materialise and update one tuple at a time is not bottlenecked by tuple size, as anticipated. Nevertheless, there is still the possibility that the tuple counts here would not be large enough to incur much of a memory bottleneck if batch processing were used.

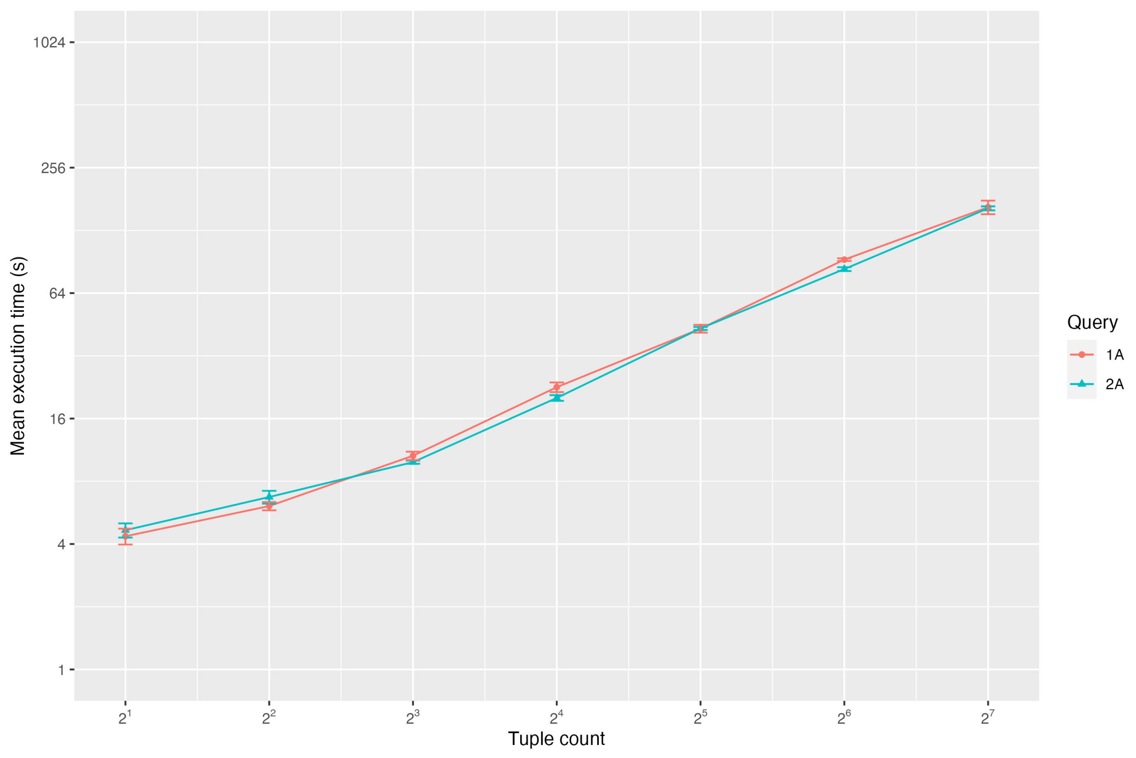


Figure 4.5: Comparison of performance of RumbleDB for Queries 1A, and 2A (Q1A, Q2A), on $\log_2$ scales

5 Conclusion

This report details the rationale and methodology behind implementing JSONiq updates in RumbleDB. Overall the project has been successful in accomplishing its main goal of displaying the viability of JSONiq updates in RumbleDB and the best practices for achieving its integration. This has been performed using a volcano-style iterator-based approach which exploits the Spark dataframes API to enable the persistence of updates via ACID-compliant Delta tables. Throughout, a variety of logical models were introduced, including an abstract decomposition for update primitives, and mutability scoping for updateable items. These models allowed us to utilise system design principles in an object-oriented paradigm to maximise extensibility without incurring severe overhead.

Based on the evaluation tests performed in this report and the test suite extended in RumbleDB's repository, updates in RumbleDB have the ability to represent a wide range and breadth of desirable alterations on highly heterogeneous and deeply nested, semi-structured data. This project has exceeded the JSONiq updates specification, identifying additional errors and semantic pitfalls to create a truly robust mechanism for describing and processing updates. Nevertheless, our experimentation also displays that RumbleDB starts with a mean execution time four times worse than SparkSQL and this linearly worsens are more data is updated, thereby indicating much room for improvement in efficiency is available.

Summatively, this project realises the combination of JSONiq's intuitive update semantics and Delta table's efficient transactional storage in RumbleDB to take another step in conquering heterogeneous Big Data. Despite the success of this implementation, it highlights numerous exciting opportunities for optimisation and the extension of what JSONiq updates mean when mapped onto other data shapes.

5.1 Future Work

Updates in RumbleDB present a variety of possible directions for extending the tool in ways that both improve efficiency and explore updating heterogeneous data as a whole.

Update primitives as they stand are limited in their scalability as they currently only target one item to update at a time. As outlined by the results of our experimentation, this approach is unsustainable for handling Big Data. Thus, it is essential to expand the suite of update primitives and applying the *Target-Selector-Content* decomposition can lead to many possibilities. One such crucial possibility is developing update primitives whose *Target* can be a collection of items in the form of an RDD or dataframe, thereby enabling batch processing of updates. However, this would entail the integration of PULs into RDD and Spark execution flows. Moreover, these update primitives acting on dataframes can then be extended to take into account the predicates applied in `where` clauses of FLWOR expressions. This model of processing would be more in line with the data-centric code generation model which is more efficient and well-suited for OLTP workloads where updates are plenty (Kersten et al., 2018).

In addition to expanding the notion of a *Target* to collections of items, it can also take into non-structured data. When persisting updates in a Delta table, the *Target* is an item of a structured data type (an object or an array) materialised before altering one component of its structure. This materialisation can incur additional overhead when taking a more batch-processing approach, especially if the object or array is particularly large. Therefore, adapting the *Target* of the update primitive to only be the atomic field being updated should eliminate unnecessary overhead. However, this approach only works when persisting to Delta tables as the *Selector* of the update primitive can be represented by the state identifying the *Target* item in the Delta table.

Finally, the set of update primitives implemented in this project lacks some useful semantics and operations for data manipulation. The most important to rectify is that no update primitive can act on collections of items that are not in an object or an array. For persisting updates in Delta tables, this has meant that rows cannot be inserted without processing the entire table as an array, inserting the rows into the array, and then reconstructing the table. Perhaps this can be implemented with built-in functions as was done in Zorba DB (zor). Additionally, certain semantics are missing, including removing a column from a structured collection (like a dataframe or Delta table) when that column is deleted for each record in the collection. An alike semantic deficiency is also present when renaming columns.

Although Delta tables are the logical option for persisting updates in RumbleDB due to the Spark API, they are not the most native option for storing heterogeneous, semi-structured data. One such native alternative is MongoDB (Chodorow and Dirolf, 2010), which natively stores and processes JSON documents, but the MongoDB API lacks the intuition and power of a query language like JSONiq. Hence, integrating MongoDB connections into RumbleDB could supply the native storage of heterogeneous data updates alongside ACID-compliant transactions, while maintaining RumbleDB's query-processing prowess. However, proper support of a system like MongoDB would be beyond the scope of just persisting updates.

Persisting updates in RumbleDB is a crucial step towards improving RumbleDB's practicality in a Big Data paradigm. Another vital step, which is briefly mentioned in section 3.5, is to extend RumbleDB with scripting capabilities. Not only would this improve the usability and intuition of RumbleDB, but it will allow for updates to be immediately used within the same script, thereby abstracting the power of the `return` clause of the transform expression to non-copied values.

A Appendix

The code for the project is found here: `https://gitlab.inf.ethz.ch/gfourny/rumble/-/tree/master-dloughlin`.

A.1 Expression Classification Enum

Code 1.1: Expression Classification Enum

```java
    public enum ExpressionClassification {
    UNSET,
    BASIC_UPDATING,
    UPDATING,
    SIMPLE,
    VACUOUS;

    public boolean isUnset() {
        return this == ExpressionClassification.UNSET;
    }

    public boolean isUpdating() {
        return this == ExpressionClassification.BASIC_UPDATING || this ==
             ExpressionClassification.UPDATING;
    }

    public boolean isSimple() {
        return !isUpdating();
    }

    public boolean isVacuous() {
        return this == ExpressionClassification.VACUOUS;
    }

    public String toString() {
        switch (this) {
            case UNSET:
                return "unset";
            case BASIC_UPDATING:
                return "basic_updating";
            case UPDATING:
                return "updating";
            case SIMPLE:
                return "simple";
            case VACUOUS:
                return "vacuous";
        }
        return null;
    }
}
```

A.2 Expression Classification Visitor Default Action

Code 1.2: Expression Classification Visitor Default Action

```java
public class ExpressionClassificationVisitor extends
    AbstractNodeVisitor<ExpressionClassification> {

@Override
protected ExpressionClassification defaultAction(Node node,
    ExpressionClassification argument) {
    ExpressionClassification expressionClassification = this.
        visitDescendants(node, argument);

    if (!(node instanceof Expression)) {
        return expressionClassification;
    }

    if (expressionClassification.isUpdating()) {
        throw new InvalidUpdatingExpressionPositionException(
                "Operand of expression is Updating when it should be
                    Simple or Vacuous",
                node.getMetadata()
        );
    }
    Expression expression = (Expression) node;
    expression.setExpressionClassification(expressionClassification);
    return expressionClassification;
}

@Override
public ExpressionClassification visitDescendants(Node node,
    ExpressionClassification argument) {
    List<ExpressionClassification> expressionClassifications = node.
        getChildren()
        .stream()
        .map(child -> this.visit(child, argument))
        .collect(Collectors.toList());
    ExpressionClassification result = expressionClassifications.
        stream()
        .anyMatch(ExpressionClassification::isUpdating)
            ? ExpressionClassification.UPDATING
            : ExpressionClassification.SIMPLE;

    return result;
}
```

A.3 Standard Update Primitive Interface

Code 1.3: Standard Update Primitive Interface

```
    public interface UpdatePrimitive {

    void apply();

    Item getTarget();

    default Item getSelector() {
        throw new UnsupportedOperationException("Operation not defined");
    }

    default Item getContent() {
        throw new UnsupportedOperationException("Operation not defined");
    }

    default List<Item> getContentList() {
        throw new UnsupportedOperationException("Operation not defined");
    }

    default boolean isDeleteObject() {
        return false;
    }

    default boolean isDeleteArray() {
        return false;
    }

    default boolean isInsertObject() {
        return false;
    }

    default boolean isInsertArray() {
        return false;
    }

    default boolean isReplaceObject() {
        return false;
    }

    default boolean isReplaceArray() {
        return false;
    }

    default boolean isRenameObject() {
        return false;
    }

}
```

A.4 Local Pending Update List Tree Map Comparator

Code 1.4: Local Pending Update List Tree Map Comparator

```
    (item1, item2) -> {
        int hashCompare = Integer.compare(item1.hashCode(), item2.
            hashCode());
        if (item1.hashCode() != item2.hashCode()) {
            return hashCompare;
        }
        if (!item1.equals(item2)) {
            return hashCompare;
        }
        return Integer.compare(
            System.identityHashCode(item1),
            System.identityHashCode(item2)
        );
```

A.5 Github Archive Schema

Code 1.5: Github Archive Schema

Schema of Github Archive dataset in dataframe notation:

```
root
 |-- actor: struct (nullable = true)
 |    |-- avatar_url: string (nullable = true)
 |    |-- gravatar_id: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- login: string (nullable = true)
 |    |-- url: string (nullable = true)
 |-- created_at: string (nullable = true)
 |-- id: string (nullable = true)
 |-- org: struct (nullable = true)
 |    |-- avatar_url: string (nullable = true)
 |    |-- gravatar_id: string (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- login: string (nullable = true)
 |    |-- url: string (nullable = true)
 |-- payload: string (nullable = true)
 |-- public: boolean (nullable = true)
 |-- repo: struct (nullable = true)
 |    |-- id: string (nullable = true)
 |    |-- name: string (nullable = true)
 |    |-- url: string (nullable = true)
 |-- type: string (nullable = true)
```

A.6 JSONiq Github Archive Delta Table Creation

Code 1.6: JSONiq Github Archive Delta Table Creation

JSONiq query that creates and validates the schema for GitHub Archive push event data where the `payload` field is a string.

```
declare type local:attempt as {
    "actor": {
        "avatar_url": "string",
        "gravatar_id": "string",
        "id": "string",
        "login": "string",
        "url": "string"
    },
    "created_at": "string",
    "id": "string",
    "org": {
        "avatar_url": "string",
        "gravatar_id": "string",
        "id": "string",
        "login": "string",
        "url": "string"
    },
    "payload": "string",
    "public": "boolean",
    "repo": {
        "id": "string",
        "name": "string",
        "url": "string"
    },
    "type": "string"
};

validate type local:attempt* {
    json-file("some/file/path/gharchive.json")
}
```

If the above is put into a JSONiq script then the data can be output to a Delta table using the command:

```
    spark-submit --packages io.delta:delta-core_2.12:2.3.0 path/to/rumbledb.jar
    run "path/to/createDeltaTableQuery.jq"
    --output-format delta
    --output-path "desired/path/to/table"
    -P 1
```

A.7 Query 1A: JSONiq

Code 1.7: Query 1A: JSONiq

```
for $d in delta-file("queries/delta_benchmark_data/newOrderTable")
return replace value of $d.NO_O_ID with ($d.NO_O_ID + 1)
```

A.8 Query 1A: SparkSQL

Code 1.8: Query 1A: SparkSQL

```
UPDATE delta.newOrderTable
SET NO_O_ID = (NO_O_ID + 1);
```

A.9 Query 1B: JSONiq

Code 1.9: Query 1B: JSONiq

```
let $h_amt := 343
for $c in delta-file("../../../queries/delta_benchmark_data/customerTable2")
return
(
    replace value of $c.C_BALANCE with $c.C_BALANCE + $h_amt,
    if ($c.C_CREDIT eq "BC")
    then
        replace value of $c.C_DATA with $c.C_DATA || $c.C_ID || $c.C_D_ID ||
                                        $c.C_W_ID || $h_amt
    else
        ()
)
```

A.10 Query 1B: SparkSQL

Code 1.10: Query 1B: SparkSQL

```
int h_amt = 343;
UPDATE delta.customerTable
SET C_BALANCE = (C_BALANCE + h_amt),
C_DATA = if(C_CREDIT == 'BC',
            concat(C_DATA, C_ID, C_D_ID, C_W_ID, h_amt),
            C_DATA
            );
```

A.11 Query 1C: JSONiq

Code 1.11: Query 1C: JSONiq

```
let $no_o_id := 4
let $w_id := 1
let $d_id_to_ol_total := (
    for $d in delta-file("../queries/delta_benchmark_data/districtTable"),
        $ol in delta-file("../queries/delta_benchmark_data/orderLineTable")
        [ $$.OL_D_ID eq $d.D_ID ]
    where $ol.OL_W_ID eq $w_id and $ol.OL_O_ID eq $no_o_id
    group by $d_id := $d.D_ID
    return { "d_id" : $d_id, "ol_total" : sum($ol.OL_AMOUNT) }
)
for $d in $d_id_to_ol_total,
    $c in delta-file("../queries/delta_benchmark_data/customerTable16")
    [ $$.C_D_ID eq $d.d_id ]
where $c.C_W_ID eq $w_id
return replace value of $c.C_BALANCE with ($c.C_BALANCE + $d.ol_total)
```

A.12 Query 1C: SparkSQL

Code 1.12: Query 1C: SparkSQL

```
int no_o_id = 4;
int w_id = 1;
WITH ol_total_per_dist AS (
    SELECT D_ID, SUM(OL_AMOUNT) AS OL_TOTAL
    FROM delta.districtTable
    JOIN delta.orderLineTable ON D_ID = OL_D_ID
    WHERE OL_W_ID = w_id AND OL_O_ID = no_o_id
    GROUP BY D_ID
)
MERGE INTO delta.customerTable USING ol_total_per_dist
ON C_D_ID = ol_total_per_dist.D_ID
WHEN MATCHED THEN
UPDATE SET C_BALANCE = (C_BALANCE ol_total_per_dist.OL_TOTAL);
```

A.13 Query 2A: JSONiq

Code 1.13: Query 2A: JSONiq

```
for $d in delta-file("../../../../queries/delta_benchmark_data/bigghTable")
```

```
    return replace value of $d.public with (not $d.public)
```

A.14 Query 2A: SparkSQL

Code 1.14: Query 2A: SparkSQL

```
UPDATE delta.bigghTable
SET public = (NOT public)
```

A.15 Query 2B: JSONiq

Code 1.15: Query 2B: JSONiq

```
for $d in delta-file("queries/delta_benchmark_data/bigghTable")
return insert { "nickname" : "cool_nickname", "popularity_rating" : -1 }
       into $d.repo
```

A.16 Query 2B: SparkSQL

Code 1.16: Query 2B: SparkSQL

```
ALTER TABLE delta.bigghTable
ADD COLUMNS (repo.nickname STRING, repo.popularity_rating INT);

UPDATE delta.bigghTable
SET repo.nickname = 'cool_nickname',
    repo.popularity_rating = -1;
```

A.17 Query 2C: JSONiq

Code 1.17: Query 2C: JSONiq

```
for $d in delta-file("queries/delta_benchmark_data/bigghTable")
return (
    insert "ids" : [
        { "repo_id" : $d.repo.id, "actor_id" : $d.actor.id, "id" : $d.id }
    ] into $d,
    delete $d.repo.id,
    delete $d.actor.id,
    delete $d.id
)
```

A.18 Query 2C: SparkSQL

Code 1.18: Query 2C: SparkSQL

```
ALTER TABLE delta.bigghTable
ADD COLUMNS (
    ids ARRAY<STRUCT<repo_id : STRING, actor_id : STRING, id : STRING>>
);

UPDATE delta.bigghTable
SET ids = array(
    named_struct('repo_id', repo.id, 'actor_id', actor.id, 'id', id)
),
repo.id = NULL,
actor.id = NULL,
id = NULL;
```

A.19 Additional Performance Graphs

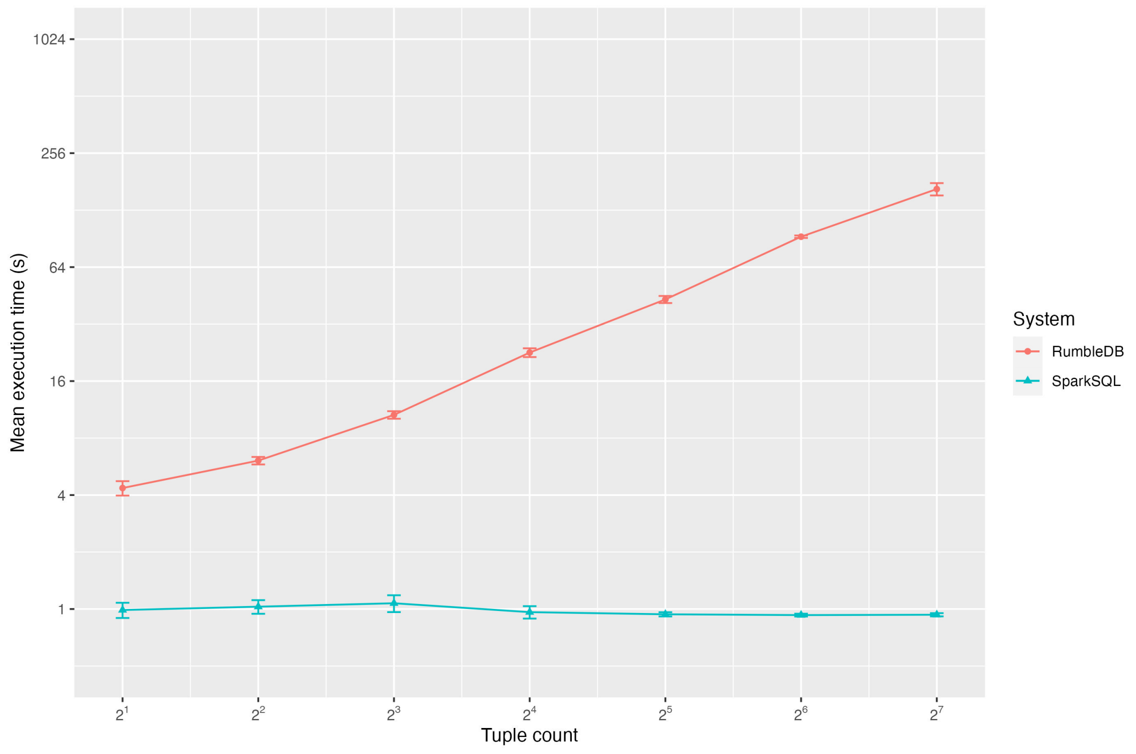


Figure A.1: Comparison of performance on Query 1A (Q1A) between RumbleDB and SparkSQL, on $\log_2$ scales

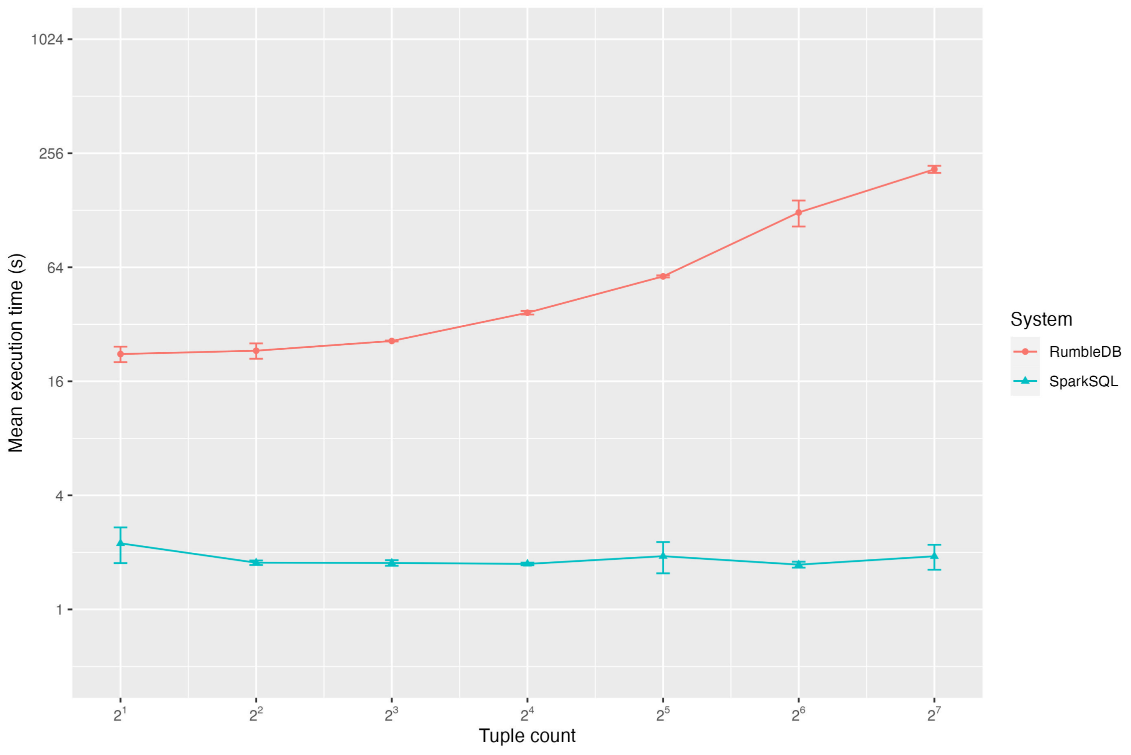


Figure A.2: Comparison of performance on Query 1C (Q1C) between RumbleDB and SparkSQL, on $\log_2$ scales – additional deficiencies over A.1 and 4.1 due to the overhead of a join in both queries

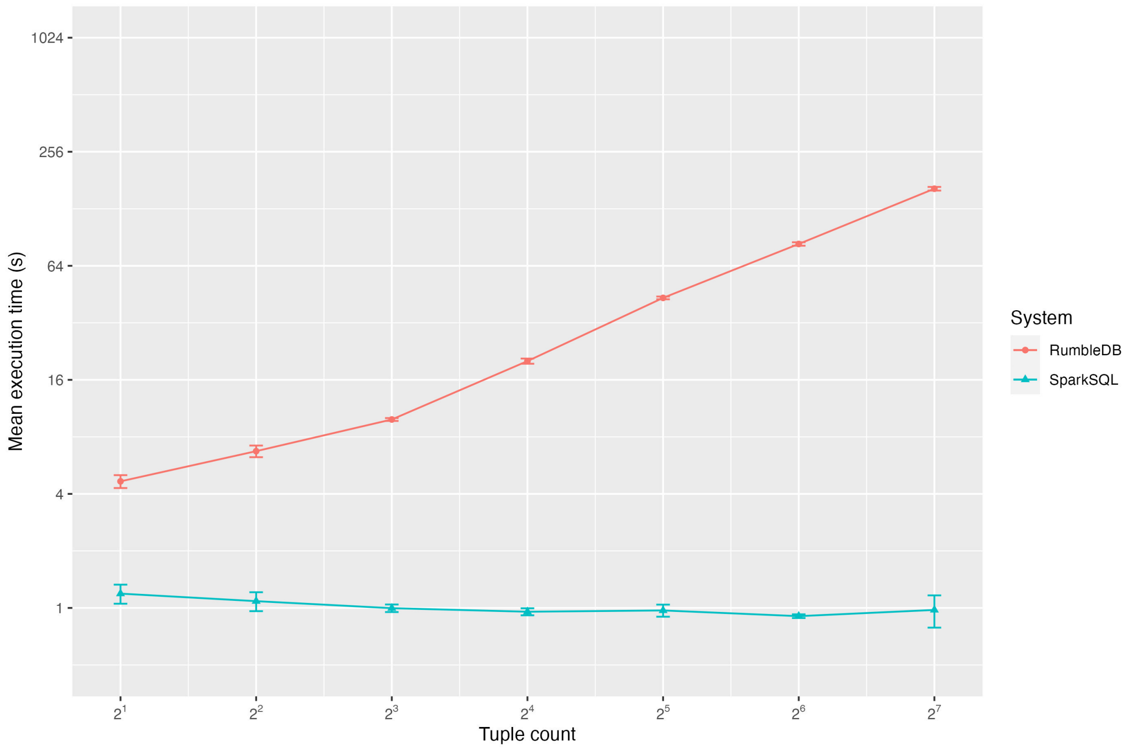


Figure A.3: Comparison of performance on Query 2A (Q2A) between RumbleDB and SparkSQL, on $\log_2$ scales

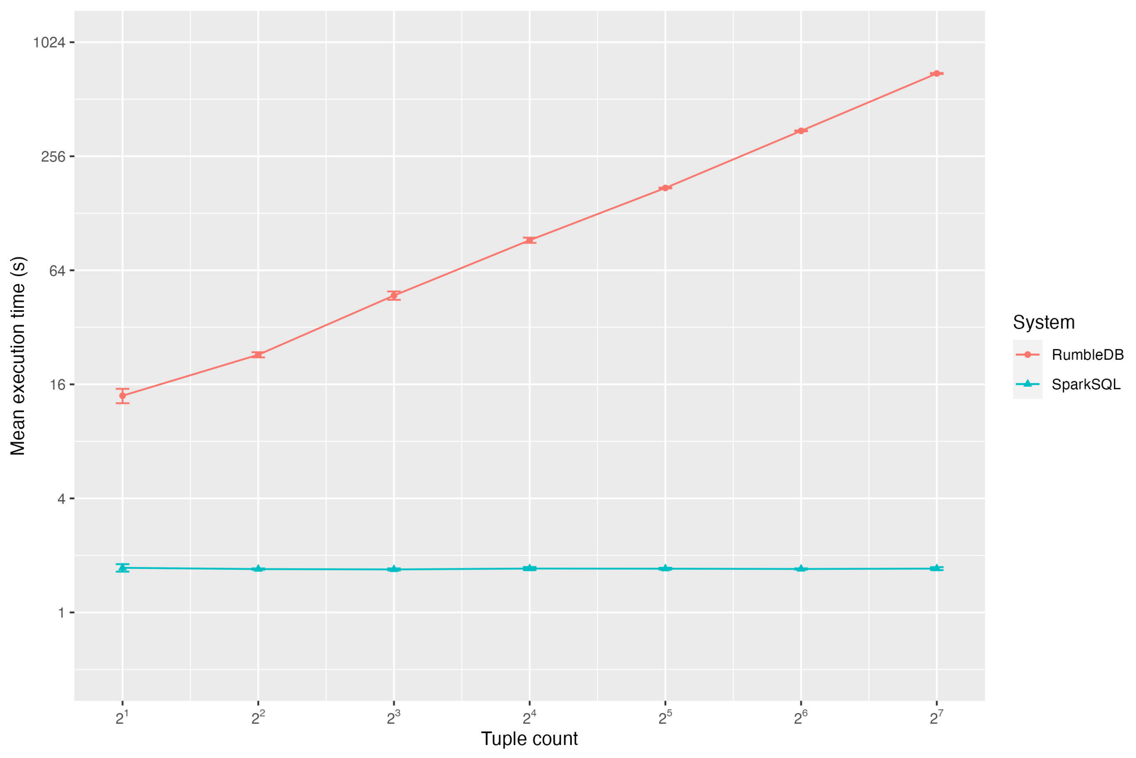


Figure A.4: Comparison of performance on Query 2C (Q2C) between RumbleDB and SparkSQL, on $\log_2$ scales

Bibliography

Apache flink - stateful computations over data streams, a. URL `https://flink.apache.org/`.

Apache kafka, b. URL `https://kafka.apache.org/`.

Apache parquet, c. URL `https://parquet.apache.org/`.

Github archive. URL `https://www.gharchive.org/`.

Mysql. URL `https://www.mysql.com/`.

Neo4j graph data platform. `https://neo4j.com/`. Accessed: 2023-02-09.

Rumbledb type validation. URL `https://rumble.readthedocs.io/en/latest/Types/`.

Zorbadb. URL `http://www.zorba.io/home`.

Amazon glacier case study, 1991. URL `https://aws.amazon.com/solutions/case-studies/snap-case-study/`.

Amazon simple storage service (s3), 2002. URL `https://aws.amazon.com/s3/`.

Amazon s3 consistency, 2002. URL `https://aws.amazon.com/s3/consistency/`.

Mariadb, Nov 2019. URL `https://mariadb.org/`.

Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.

J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide: time to relax.* " O'Reilly Media, Inc.", 2010.

Apache Commons Documentation Apache Commons Team. Apache commons. URL `https://commons.apache.org/`.

Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. Delta lake: high-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424, 2020.

Narsimha Banothu, ShankarNayak Bhukya, and K Venkatesh Sharma. Big-data: Acid versus base for database transactions. In *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT)*, pages 3704–3709. IEEE, 2016.

Kevin S Beyer, Vuk Ercegovac, Rainer Gemulla, Andrey Balmin, Mohamed Eltabakh, Carl-Christian Kanne, Fatma Ozcan, and Eugene J Shekita. Jaql: A scripting language for large scale semistructured data analysis. *Proceedings of the VLDB Endowment*, 4(12):1272–1283, 2011.

Paul V Biron, Ashok Malhotra, World Wide Web Consortium, et al. Xml schema part 2: Datatypes, 2004.

Scott Boag, Don Chamberlin, Mary F Fernández, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. Xquery 1.0: An xml query language. 2002.

Dhruba Borthakur et al. Hdfs architecture guide. *Hadoop apache project*, 53(1-13):2, 2008.

Zouhaier Brahmia, Safa Brahmia, Fabio Grandi, and Rafik Bouaziz. Jupdate: a json update language. *Electronics*, 11(4):508, 2022.

Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0, 1998.

Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

Francesco Cappa, Raffaele Oriani, Enzo Peruffo, and Ian McCarthy. Big data for creating and capturing value in the digitalized environment: unpacking the effects of volume, variety, and veracity on firm performance. *Journal of Product Innovation Management*, 38(1):49–67, 2021.

Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An xml query language for heterogeneous data sources. In *International Workshop on the World Wide Web and Databases*, pages 1–25. Springer, 2000.

Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010. ISBN 1449381561.

Laura Clarke, Xiangqun Zheng-Bradley, Richard Smith, Eugene Kulesha, Chunlin Xiao, Iliana Toneva, Brendan Vaughan, Don Preuss, Rasko Leinonen, Martin Shumway, et al. The 1000 genomes project: data management and community access. *Nature methods*, 9(5):459–462, 2012.

Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

World Wide Web Consortium et al. Xml information set. 2004.

World Wide Web Consortium et al., 2011. URL `https://www.w3.org/TR/xquery-update-10/`.

World Wide Web Consortium et al., 2017. URL `https://www.w3.org/TR/xpath-datamodel-31/`.

George P Copeland and Setrag N Khoshafian. A decomposition storage model. *Acm Sigmod Record*, 14(4):268–279, 1985.

TPC Council. C homepage. URL `https://www.tpc.org/tpcc/`.

Douglas Crockford. The application/json media type for javascript object notation (json). Technical report, 2006.

Alfredo Cuzzocrea. Big data lakes: models, frameworks, and techniques. In *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 1–4. IEEE, 2021.

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

Sérgio Fernandes and Jorge Bernardino. What is bigquery? In *Proceedings of the 19th International Database Engineering & Applications Symposium*, pages 202–203, 2015.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445, 2018.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)*, 25 (2):73–169, 1993.

PostgreSQL Global Development Group. Postgresql. URL `https://www.postgresql.org/`.

Dick Grune and Ceriel JH Jacobs. Parsing techniques (monographs in computer science), 2006.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proceedings of the VLDB Endowment*, 11(13):2209–2222, 2018.

Doug Laney et al. 3d data management: Controlling data volume, velocity and variety. *META group research note*, 6(70):1, 2001.

Volker Markl. Breaking the chains: On declarative data analysis and data independence in the big data era. *Proceedings of the VLDB Endowment*, 7(13):1730–1733, 2014.

ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

Jan Motl and Oliver Schulte. The ctu prague relational learning repository, 2015.

Ingo Müller, Ghislain Fourny, Stefan Irimescu, Can Berker Cikis, and Gustavo Alonso. Rumble: data independence for large messy data sets. *Proceedings of the VLDB Endowment*, 14(4):498–506, 2020.

Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. The sql++ unifying semi-structured query language, and an expressiveness benchmark of sql-on-hadoop, nosql and newsql databases. *CoRR, abs/1405.3631*, 2014.

Jens Palsberg and C Barry Jay. The essence of the visitor pattern. In *Proceedings. The Twenty-Second Annual International Computer Software and Applications Conference (Compsac'98)(Cat. No. 98CB 36241)*, pages 9–15. IEEE, 1998.

Terence Parr. The definitive antlr 4 reference. *The Definitive ANTLR 4 Reference*, pages 1–326, 2013.

Dan Pritchett. Base: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability. *Queue*, 6(3):48–55, 2008.

Dave Raggett, Arnaud Le Hors, Ian Jacobs, et al. Html 4.01 specification. *W3C recommendation*, 24, 1999.

Jonathan Robie, Ghislain Fourny, Matthias Brantner, Daniela Florescu, Till Westmann, and Markos Zaharioudakis. Language specification 0.4 jsoniq. 2011.

Alex Rodriguez. Restful web services: The basics. *IBM developerWorks*, 33(2008):18, 2008.

Seref Sagiroglu and Duygu Sinanc. Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)*, pages 42–47. IEEE, 2013.

Pegdwendé Sawadogo and Jérôme Darmont. On data lake architectures and metadata management. *Journal of Intelligent Information Systems*, 56:97–120, 2021.

Odysseas G Tsatalos, Marvin H Solomon, and Yannis E Ioannidis. The gmap: A versatile tool for physical data independence. *The VLDB Journal*, 5:101–118, 1996.

Tom White. Hadoop: The definitive guide, 2015; published by oreilly media.

Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.