Building a JSONiq Query Optimizer using MLIR

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Bachelor's Thesis Nr. 338b

Systems Group, Department of Computer Science, ETH Zurich

Building a JSONiq Query Optimizer using MLIR

by

Martin Fiebig

Supervised by

Prof. Gustavo Alonso, Dr. Ghislain Fourny, Ingo Müller

October 2020 – December 2020

D INFK

Abstract

Semi-structured data formats like JSON gained popularity through their ability to represent arbitrarily complex data in a way that it can easily be read and written by humans, and parsed and generated by machines. This simplicity is especially useful for applications where it is not worth to spend time in schema design and data migration. However, it comes at a price: Query execution is much slower.
In this bachelor's thesis we apply some optimizations on a MLIR dialect for JSONiq. We also take a closer look at type inference for a selection of JSONiq expressions.

Contents

1 Introduction **3**

2 Background and Related Work **3**
2.1 JSON 4
2.2 JSONiq 4
2.3 Rumble 5
2.4 MLIR 6

3 The JSONiq Dialect **7**

4 Type Inference **10**
4.1 Motivation 10
4.2 Algorithm 10
4.3 Implementation 11
4.4 Inference Rules 12

5 Optimizations **12**
5.1 Short circuit evaluation 12
5.2 Logical identity 13
5.3 Testing 13

6 Conclusion and Future Work **14**

1 Introduction

Semi-structured data formats like JSON gained popularity through their ability to represent arbitrarily complex data in a way that it can easily be read and written by humans, and parsed and generated by machines. This simplicity is especially useful for applications where it is not worth to spend time in schema design and data migration. However, it comes at a price: Query execution is much slower.
In this thesis, we apply some optimizations on JSONiq queries using the MLIR compiler infrastructure. The query is translated into an extended version of a previously defined MLIR dialect [11] and then, using operation transformations, optimized and translated back to an equivalent JSONiq query. Figure 1 is an illustration of the pipeline.

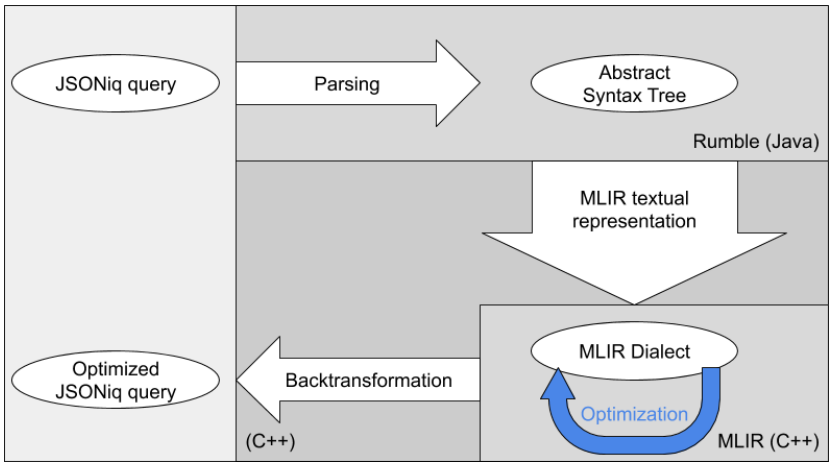


Figure 1: Illustration of the optimization pipeline. The part in blue is the scope of this thesis.

First, we introduce some background and related work, including the MLIR dialect and our modifications to it in chapter 3. Chapter 4 introduces the idea and motivation of type inference. It also discusses, how this helps improving query efficiency. Some optimizations together with their up- and downsides are presented in chapter 5. Finally chapter 6 discusses further work, which we could not look at due to time constraints.

2 Background and Related Work

JSONiq queries manipulate data in JSON format. They are executed with a JSONiq engine such as Rumble. In the following, these concepts as well as the basics of the MLIR compiler infrastructure are introduced.

2.1 JSON

JSON [2] (*JacaScript Object Notation*) provides a textual representation of data, which can be easily understood by both, humans and machines. It uses conventions familiar to many programmers of various programming languages, and is therefore widely used for data-interchange.
The basic structures are unordered sets of name-value pairs, called objects, and ordered collections of values, called arrays. This two concepts are present in many programming languages as different concepts, like records, structs and dictionaries, and vectors and list, respectively.
Values can be strings, numbers, booleans or nulls, and even objects or arrays, which allows for nested structures.

```
{ “Name” : “Boris”,
  “Matriculation number” : “14-023-486”,
  “Address” : {“Street” : “13 Monday Street”, “Postal code” : “29842”, “City” : “Urbantown”},
  “Entry date” : 2018-09-19,
  “Courses” : [“Compiler Design”, “Parallel Programming”, “Databases”],
  “Grades” : [5.0, 4.5, 4.75] }
```

Figure 2: Example of a JSON object

2.2 JSONiq

JSONiq [3] is a functional and declarative language for processing and querying nested, heterogeneous, and semi-structured JSON data. The main building blocks are expressions, which are defined recursively and can therefore be expressed by expression trees.
Formally, expressions are mappings from a JSONiq tuple to a sequence of JSONiq items: $expr \in E : T \rightarrow S$, where E is the set of expressions, T the set of tuples and S is the set of sequences of items. The input tuple is unmodified pushed down to the child expressions, which return their output as a sequence of items. The expression itself computes its output sequence out of the results of the child expressions. Figure 3 illustrates such an expression tree.

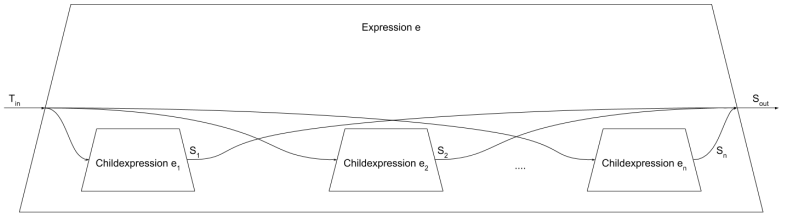


Figure 3: Illustration of a general expression tree

As the name suggest, a sequence is an ordered list of items. JSONiq items are all JSON values and many other atomics such as dates, durations and binaries.

As a special case, a singleton sequence is defined to be the same as the only item of the sequence.
JSONiq tuples are collections of key-value pairs, where each key represent a variable and its value corresponds to the associated sequence of items. These tuples can be grouped together to form a tuplestream, a vector of tuples. Figure 4 presents a tuplestream consisting of 3 tuples. The first tuple contains 5 keys, where e.g. *$y* is associated with a sequence of 4 items.

```
<$x: (true, ["hello", "world"]), $y: (3e5, false, null, "foo"), $z: ({"bar": 42, "foo": true), $min: (1), $max: (10)>
<$x: (), $z: (true, null)>
<$a: ("foo")>
```

Figure 4: Example of an JSONiq tuplestream

The most important expression in JSONiq is the FLWOR expression. It may consist of multiple clauses in any order out of the seven available ones: For, Let, Count, Group by, Order by, Where and Return. Every FLWOR expression must begin with either a For clause or a Let clause, and end with a Return clause. In contrast to expressions, clauses are mapping from tuple streams to tuple streams: $clause \in C : TS \rightarrow TS$, where C is the set of clauses and TS is the set of tuple streams. An exception is the Return clause, which maps a tuple stream to a sequence of items: $return : TS \rightarrow S$. Since every expression takes as input a tuple, the FLWOR expression computes the tuplestream consisting only of its input tuple as input to the first clause. Figure 5 shows the expression tree for the FLWOR expression.

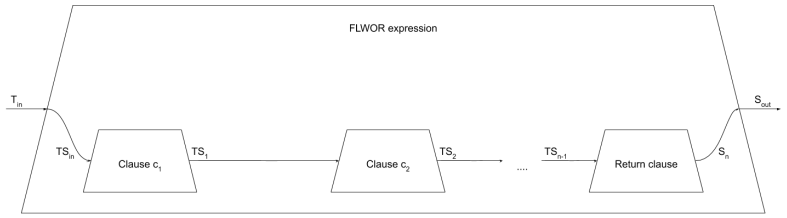


Figure 5: Illustration of the expression tree in case of the FLWOR expression. TS_{in} is the input tuple stream consisting of the input tuple T_{in} only.

2.3 Rumble

Rumble [7, 10] is a JSONiq engine built on top of Apache Spark [1, 12], a popular and fast cluster computing system. It processes large, heterogeneous and nested collections of JSON objects by dynamically pushing down computations to Spark, without exposing this to the user [5]. The query is translated into an abstract syntax tree (AST) which, in turn, is transformed into a tree of runtime iterators.

Rumble switches dynamically between three execution modes: For small amount of data, and pre- and postprocessing spark jobs, the queries can be executed locally. Simple queries can run entirely in this mode. If there is no knowledge about the structure of the data, a RDD-based execution is chosen. Resilient Distribued Datasets (RDD) is a structure used in Spark to process flat collections of hetereogeneous data. In case, parts of the structure are statically known, the engine switches to a DataFrame-based execution mode. DataFrames is another structure in Spark. It is used for collections of homogeneous rows whose type and field names are statically known.

2.4 MLIR

MLIR (Multi-Level Intermediate Representation) [4,9] is a flexible and extensible infrastructure for compiler construction. It is based on a graph like structure where nodes represent operations and edges represent values passed between operations.

Dialects
The mechanism by which the MLIR ecosystem can be extended, are dialects. They are identified by a unique string and form groups for operations, attributes and types which are semantically connected.

Operations
The main building blocks are operations. Operations take and return zero or more operands and results, respectively. They are fully extensible and allow to "represent many different concepts, from higher-level concepts like function definitions, function calls, buffer allocations, view or slices of buffers, and process creation, to lower-level concepts like target-independent arithmetic, target-specific instructions, configuration registers, and logic gates." [4] Each operation may contain zero or more regions. The control flow between these regions is defined by the operation itself. A region is an ordered list of blocks, where the entry block must not be a succesor to any other block. Blocks are ordered lists of operations representing sequential execution. They can take input arguments and attributes, and need to be terminated by a terminator operation, a special instance of an operation. Further special instances are module operations, which consist of only one region and one block, and function operations, which consists of only one region representing the function body.

Declarative Framework
MLIR also introduces a declarative framework. The Operation Definition Specification (ODS) allows to define operations and their verifiers and properties declaratively. The TableGen-based [8] definitions are translated to C++ code removing the burden of boiler plate code from the dialect designer.
Transformations of operations can often be expressed as simple transformations on the directed acyclic graph (DAG) defined by the relation of SSA values.

These can be declaratively defined with the help of Declarative Rewrite Rules (DRR).

3 The JSONiq Dialect

To optimize JSONiq queries, we need to express them in MLIR. There is already a JSONiq dialect for MLIR designed for this purpose [11]. The internal AST in Rumble is translated to a textual representaion, which can be parsed by the MLIR infrastructure using the JSONiq dialect.

In this dialect, each JSONiq expression (except for the FLWOR expression) is represented by exactly one operation. Unary expressions are mapped to operations taking one input operand and returning one result. Operations with zero, two, three or more input operands are representations of nullary, binary, ternary and n-ary expressions, respectively. JSONiq typing expressions are unary expression with additional information about the type to consider. They are represented the same way as unary expressions with an aditional input attribute for the type information. Figure 6 pictures the RangeExpression as a simple example. The first two lines are IntegerLiteralExpressions for the literals `0` and `10`. They are stored in the registers 1 and 2, respectively. In the third line, we find the actual RangeExpression, which takes the two literals stored in the registers 1 and 2 as input. Its output is stored in register 3. Together, the three lines represent the expression `0 to 10`. A complete list of the mapping from JSONiq expressions to MLIR operations is shown in Table 1.

```
%1 = "jsoniq.lit"() {value = 0 : i64 } : () -> !jsoniq.sequence
%2 = "jsoniq.lit"() {value = 10 : i64 } : () -> !jsoniq.sequence
%3 = "jsoniq.to"(%1, %2) : (!jsoniq.sequence, !jsoniq.sequence) -> !jsoniq.sequence
```

Figure 6: A JSONiq range expression in the MLIR dialect

The PredicateExpression is the only case where the operation has a region attached to it. It represents the expression used to compute the boolean value of the item. An example of the PredicateExpression is pictured in Figure 7.

The different LiteralExpressions (for each JSONiq atomic one) are mapped to the same operations, where the value of the literal is saved as a string attribute. To avoid ambiguity between the non-string literals and their string representation, we use MLIR's ability to determine the type of the attribute at parsing time. This way, we can differentiate between the LiteralExpressions by looking at the attribute type. Table 2 shows some examples of Literal-

Literal Expressions	
IntegerLiteralExpression	$jsoniq.lit\,\{value : i64\} : () \rightarrow !jsoniq.sequence$
StringLiteralExpression	$jsoniq.lit\,\{value : String\} : () \rightarrow !jsoniq.sequence$
DoubleLiteralExpression	$jsoniq.lit\,\{value : f64\} : () \rightarrow !jsoniq.sequence$
DecimalLiteralExpression	$jsoniq.lit\,\{value : f64\} : () \rightarrow !jsoniq.sequence$
NullLiteralExpression	$jsoniq.lit\,\{value\} : () \rightarrow !jsoniq.sequence$
BooleanLiteralExpression	$jsoniq.lit\,\{value : i1\} : () \rightarrow !jsoniq.sequence$
Unary Expressions	
NotExpression	$jsoniq.not : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
NegExpression	$jsoniq.neg : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
ArrayConstructorExpression	$jsoniq.arrayconstructor : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
ArrayUnboxingExpression	$jsoniq.arrayunboxing : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
Binary Expressions	
AdditiveExpression	$jsoniq.+ : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
	$jsoniq.- : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
RangeExpression	$jsoniq.to : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
ComparisonExpression	$jsoniq.*cmp* : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
MultiplicativeExpression	$jsoniq.*op* : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
AndExpression	$jsoniq.and : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
OrExpression	$jsoniq.or : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
StringConcatExpression	$jsoniq.\vert\vert : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
ArrayLookupExpression	$jsoniq.[[]] : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
ObjectLookupExpression	$jsoniq.objectlookup : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
Ternary Expressions	
ConditionalExpression	$jsoniq.conditional : (!jsoniq.sequence, !jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
N-ary Expressions	
FunctionCallExpression	$jsoniq.func\,\{funcname : string\} : (!jsoniq.sequence)^N \rightarrow !jsoniq.sequence$
Typing Expressions	
TreatExpression	$jsoniq.treat\,\{type : string\} : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
CastExpression	$jsoniq.cast\,\{type : string\} : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
InstanceOfExpression	$jsoniq.instanceof\,\{type : string\} : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
CastableExpression	$jsoniq.castable\,\{type : string\} : (!jsoniq.sequence) \rightarrow !jsoniq.sequence$
Special Expressions	
VariableReferenceExpression	$jsoniq.varref\,\{var : string\} : (!jsoniq.tuple) \rightarrow !jsoniq.sequence$
ObjectConstructorExpression	$jsoniq.constructobject : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
	$jsoniq.mergeobjects : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
	$jsoniq.emptyobject : () \rightarrow !jsoniq.sequence$
CommaExpression	$jsoniq.comma : (!jsoniq.sequence, !jsoniq.sequence) \rightarrow !jsoniq.sequence$
PredicateExpression	$jsoniq.[]$(^bb0(%arg0 :!$jsoniq.sequence$) : //body }) $\{var : string\} : (!jsoniq.sequence) \rightarrow !jsoniq.seuqence$

Table 1: Mapping from JSONiq expressions to MLIR operations

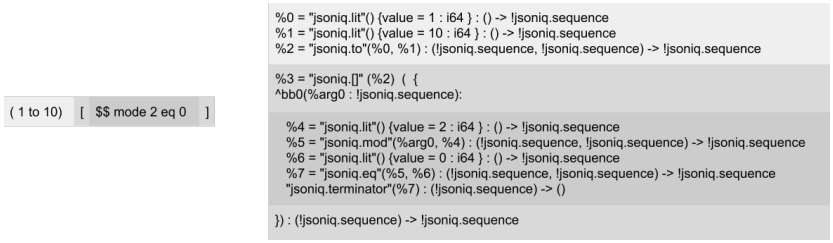


Figure 7: A JSONiq predicate expression in the JSONiq dialect

2020	$jsoniq.lit\,\{value = 2020 : i64\} : () \rightarrow !jsoniq.sequence$	$jsoniq.lit\,\{value = 2020 : i64\} : () \rightarrow !jsoniq.sequence$
true	$jsoniq.lit\,\{value = "true"\} : () \rightarrow !jsoniq.sequence$	$jsoniq.lit\,\{value = true : i1\} : () \rightarrow !jsoniq.sequence$
"true"	$jsoniq.lit\,\{value = "true"\} : () \rightarrow !jsoniq.sequence$	$jsoniq.lit\,\{value = "true"\} : () \rightarrow !jsoniq.sequence$
null	$jsoniq.lit\,\{value = "null"\} : () \rightarrow !jsoniq.sequence$	$jsoniq.lit\,\{value\} : () \rightarrow !jsoniq.sequence$
"null"	$jsoniq.lit\,\{value = "null"\} : () \rightarrow !jsoniq.sequence$	$jsoniq.lit\,\{value = "null"\} : () \rightarrow !jsoniq.sequence$

Table 2: Literal expressions in JSONiq (on the left), with their representations in the original dialect (in the middle) and the extended dialect (on the right)

Expressions in JSONiq, the original dialect, and our extended version of the dialect.

All inputs and outputs of the mentioned operations above are of type JSONiq sequence. To enable some optimizations, we extend the type system by adding further information to the sequence type. The parameters *minLength* and *maxLength*, inform us about how many items are in the sequence at least and at most, respectively. A special value denotes the case where there are not any statically known information about the minimal and maximal number of items. In the case of homogeneous sequences, an additional parameter *elementType* gives us the type of every element of the item. Heterogeneous sequences are currently treated as homogeneous sequences with every element of type `item`. We also add some atomic types like booleans, integers and strings. This extension allows us to easily determine special sequences: the empty sequence where $minLength = maxLength = 0$, the singleton sequence where $minLength = maxLength = 0$, the sequence of at most one item ($maxLength = 1$), the sequence of at least one item ($minLength = 1$), and the sequence of zero or one item ($minLength = 0$ and $maxLength = 1$).

The FLWOR expression is mapped to multiple operations, for each clause one operation. Their one input and output is a tuplestream, except for the Return operation, which returns a sequence of items. Most of these operations have a region attached to it, which represents the body of the clause. A sequential grouping of these clause operations in a block represents a FLWOR expression. The mappings can be found in Table 3 and an example of a FLWOR expression is included in Figure 8.

A query itself is a function operation with one region and block. It takes

ForClause	$jsoniq.for(\{\hat{}bb0(\%arg0 : !jsoniq.tuple) :$ $//body$ $\}) \{var : string\} : (!jsoniq.tuplestream) \rightarrow !jsoniq.tuplestream$
LetClause	$jsoniq.let(\{\hat{}bb0(\%arg0 : !jsoniq.tuple) :$ $//body$ $\}) \{var : string\} : (!jsoniq.tuplestream) \rightarrow !jsoniq.tuplestream$
WhereClause	$jsoniq.where(\{\hat{}bb0(\%arg0 : !jsoniq.tuple) :$ $//body$ $\}) : (!jsoniq.tuplestream) \rightarrow !jsoniq.tuplestream$
CountClause	$jsoniq.count\{var : string\} : (!jsoniq.tuplestream) \rightarrow !jsoniq.tuplestream$
OrderByClause	$jsoniq.orderby(\{\hat{}bb0(\%arg0 : !jsoniq.tuple) :$ $//body$ $\}) \{rule : string\} : (!jsoniq.tuplestream) \rightarrow !jsoniq.tuplestream$
GroupByClause	$jsoniq.groupby(\{\hat{}bb0(\%arg0 : !jsoniq.tuple) :$ $//body$ $\}) \{var : string\} : (!jsoniq.tuplestream) \rightarrow !jsoniq.tuplestream$
ReturnClause	$jsoniq.return(\{\hat{}bb0(\%arg0 : !jsoniq.tuple) :$ $//body$ $\}) : (!jsoniq.tuplestream) \rightarrow !jsoniq.sequence$

Table 3: Mapping from JSONiq clauses to MLIR operations

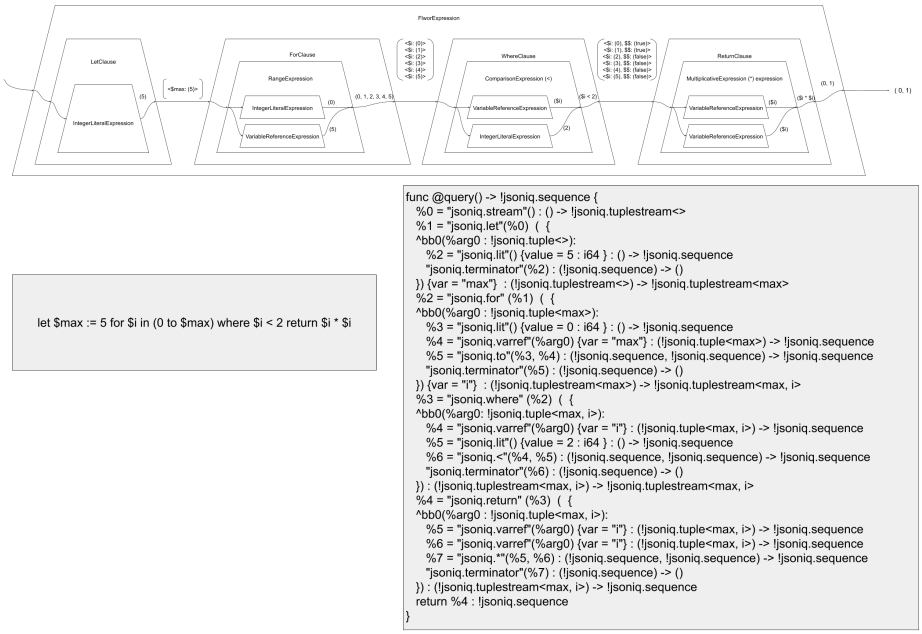
no input and returns a sequence of items. The body contains the operations corresponding to the expression tree. An illustration of an expression tree and its representation in the JSONiq dialect is shown in figure 8.

4 Type Inference

4.1 Motivation

JSONiq is a strongly typed programming language, but allows to ommit type annotations. In such a case, the most general type `item*` is assumed. For variables of this type, information like the number of items, the actual type of each item and which properties we do not know need to be stored. In contrast the type `double` is a 64-bit width word as defined by IEEE. Infering the type of these expressions can lead to more efficient queries with less memory usage. It allows to reduce type checks and points to dead code. An example is the `cast` expression. If we have, for instance, the expression `$x cast as boolean` [1] and can infere the type of `x` to be `boolean`, then this expression can be replaced by `$x`. Another example is the expression `$x + $y`. It requires both, `x` and `y` to be a numeric type (`integer`, `decimal`, `double`, etc.). Without any type annotations, the type of the variables need to be checked at runtime. Knowing statically, that both variable references are of the same numeric type, allows to ommit these type checks.

[1] We assume that variables are defined either globally in the prolog or locally shortly before the expression.




```
let $max := 5 for $i in (0 to $max) where $i < 2 return $i * $i
```

```
func @query() -> !jsoniq.sequence {
  %0 = "jsoniq.stream"() : () -> !jsoniq.tuplestream<>
  %1 = "jsoniq.let"(%0)  (  {
  ^bb0(%arg0 : !jsoniq.tuple<>):
    %2 = "jsoniq.lit"() {value = 5 : i64 } : () -> !jsoniq.sequence
    "jsoniq.terminator"(%2) : (!jsoniq.sequence) -> ()
  }) {var = "max"}  : (!jsoniq.tuplestream<>) -> !jsoniq.tuplestream<max>
  %2 = "jsoniq.for" (%1)  (  {
  ^bb0(%arg0 : !jsoniq.tuple<max>):
    %3 = "jsoniq.lit"() {value = 0 : i64 } : () -> !jsoniq.sequence
    %4 = "jsoniq.varref"(%arg0) {var = "max"} : (!jsoniq.tuple<max>) -> !jsoniq.sequence
    %5 = "jsoniq.to"(%3, %4) : (!jsoniq.sequence, !jsoniq.sequence) -> !jsoniq.sequence
    "jsoniq.terminator"(%5) : (!jsoniq.sequence) -> ()
  }) {var = "i"}  : (!jsoniq.tuplestream<max>) -> !jsoniq.tuplestream<max, i>
  %3 = "jsoniq.where" (%2)  (  {
  ^bb0(%arg0: !jsoniq.tuple<max, i>):
    %4 = "jsoniq.varref"(%arg0) {var = "i"} : (!jsoniq.tuple<max, i>) -> !jsoniq.sequence
    %5 = "jsoniq.lit"() {value = 2 : i64 } : () -> !jsoniq.sequence
    %6 = "jsoniq.<"(%4, %5) : (!jsoniq.sequence, !jsoniq.sequence) -> !jsoniq.sequence
    "jsoniq.terminator"(%6) : (!jsoniq.sequence) -> ()
  }) : (!jsoniq.tuplestream<max, i>) -> !jsoniq.tuplestream<max, i>
  %4 = "jsoniq.return" (%3)  (  {
  ^bb0(%arg0 : !jsoniq.tuple<max, i>):
    %5 = "jsoniq.varref"(%arg0) {var = "i"} : (!jsoniq.tuple<max, i>) -> !jsoniq.sequence
    %6 = "jsoniq.varref"(%arg0) {var = "i"} : (!jsoniq.tuple<max, i>) -> !jsoniq.sequence
    %7 = "jsoniq.*"(%5, %6) : (!jsoniq.sequence, !jsoniq.sequence) -> !jsoniq.sequence
    "jsoniq.terminator"(%7) : (!jsoniq.sequence) -> ()
  }) : (!jsoniq.tuplestream<max, i>) -> !jsoniq.sequence
  return %4 : !jsoniq.sequence
}
```

Figure 8: A JSONiq query (bottom left) in the JSONiq dialect (bottom right) and it's expression tree (top)

4.2 Algorithm

To infere the types of the results of the operations, we build a worklist consisting of all operations which return a general type. We defined a general type to be every type which is not the empty sequence or a singleton sequence with element type other than `item`. In this worklist, we find an operation with no operands of a general type. We remove this operation from the worklist and infer its type. In case, we could not find an operation, we break out of the loop. Since we remove an operation in every iteration, we will reach a state where the worklist is either empty or contains only operations with at least one operand of a general type. At this point we do not find another operation and break out of the loop. Therefore the algorithm terminates. However, it does infer the type only for operations where all input operands are already known to be of a specific type. Consider the comparison expressions. Independently of the input arguments, these expressions either throw an error due to incomparability of the operands, or the result will be of type `boolean`. To account for operations like this, we would need to iterate over all operations in the worklist and infer their result types until we reach a fix point.

LiteralExpression	The result type is determined from the attribute type.
ComparisonExpression	The result type is always `boolean`.
AndExpression OrExpression NotExpression	The result type is always `boolean`.
MultiplikativeExpression AdditiveExpression	The result type is the most general numeric type of the operands.
ConditionalExpression	If we can determine the type of the child expressions, and if they are the same, then the result type will also be the same.
CastableExpression InstanceofExpression	The result type is always `boolean`.
CastExpression TreatExpression	The result type is determined from the value of the string attribute.

Table 4: Type Inference Rules for a selection of expressions

4.3 Implementation

We implemented type inference as a custom pass in MLIR. It is a class *TypeInference* inheriting from *mlir::PassWrapper* and overriding the method *runOnFunction*. Our algorithm from section 4.1 is implemented there. We also define a custom MLIR interface *TypeInferenceInterface*. In MLIR, Interfaces are used to ensure the implementation of specific functions within operations. In our case it is the method *inferTypes()* which is called within the type inference pass. Finally, the pass is registered in a pass manager which will apply it from now on.

4.4 Inference Rules

The inference rule for the LiteralExpression is straightforward. Based on the type of the attribute *value* we can determine the result type. In case of an integer attribute, the result will be of type `integer`, in case of a boolean attribute, the result will be of type `boolean`. For comparison and logical exressions, the result type will always be `boolean`, or they return an error. Type inference is also possible if we have a conditional expression (`if` *e1* `then` *e2* `else`). If we can determine the types of both expressions, *e1* and *e2* to be the same, then the return type of the conditional expression will also be the same. Table 4 lists a few more rules.

5 Optimizations

We implemented three optimizations: Short circuit evaluation, Logical identity and Double Not.

5.1 Short circuit evaluation

JSONiq does not define the order of evaluation for operands [3]. For instance the expression `true or (1 div 0)` may throw an error or return `true`. It follows that any execution engine can choose which operand it wants to evaluate first. We make use of it and apply the short circuit evaluation from both sides of the operation. This means that for every expression `true or` e where e is some subexpression, we replace it by `true`. We do the same for e `or true`. For every expression `false and` e and e `and false`, we replace it by `false`. This transformation is correct due to the semantics of the logical operators AND and OR. It defines that when ever we have TRUE OR *something* it is always TRUE. Since OR is commutative, it also holds the other way around. The semantics of the AND operator is defined in a similar way.

5.2 Logical identity

The semantics of the operators AND and OR also define some kind of identiy. This means that FALSE OR *something* and TRUE OR *something* can be replaced by *something* as long as it is a boolean value. Applied to JSONiq, we conclude that every expression `true or` e and `false and` e, where e is some subexpression, can be replaced by e `cast as boolean`. We need to cast the subexpression e to `boolean`, since the operands of the operations `and` and `or` need to be of type `boolean`. Normally, this cast is implicitly done by the execution engine, but since we optimize the operation away, we need to add an explicit cast. Again, `and` and `or` are commutative, so the same holds for e `or true` and e `and false`.

Similar, we conclude the transformation rule for `not (not` e `)`. We can replace it by e `cast as boolean`.

5.3 Testing

Our optimizations should not change the output or semantics of the original query. To ensure this, we used the runtime tests found in the gitlab repository of Rumble [6]. First, we modified the test suite to produce the textual representation of the query in the MLIR dialect and write it to a file. We than run our optimizer over the generated files and backtransformed the result to a JSONiq query. Finally, we run the test suite again, but this time with the optimized queries as input. Since the test suite is also checking the correctness of the parser in Rumble, not all original queries will be translated to MLIR. The queries with expected parsing errors will and should be excluded for testing the optimizations due to the fact that the optimization step will never be reached. Nontheless, we found some problems in generating the MLIR IR. Corner cases, where the ArrayConstructorExpression and the CommaExpression receive less than two operands are not handled in the right way.
Further problems occur when parsing the generated files and back transforming the MLIR IR to a JSONiq query. The MLIR dialect misses definitions for constructs found in the generated textual representation, leading to parsing errors.

Rumble parsing errors (no MLIR generated)	18
MLIR generation errors	215
MLIR parsing errors	18
Change in metadata and errors in backtransformation	63
overall	**314**

Table 5: Absolute number of errors

An example of such a construct is the type `tuplestream`.
During backtransformation, we make use of our own output format. This leads to problems with the metadata used to provide location information about the expected error. There are also special characters in string values which are handled differently in Rumble and MLIR (C++). The absolute number of errors can be found in Table 5. Out of 709 test cases only 395 passed the tests.

6 Conclusion and Future Work

In this thesis, we discussed type inference and some optimizations for a selection of simple JSONiq expressions. We did not consider all expressions and clauses. For example, LetClause folding or merging multiple WhereClauses into one is interesting to look into.
We also could not get a look at experimental evaluation with respect to the efficiency of query execution. Here we need to consider that the optimizations are done after the query is submitted, shortly before the execution. To the user, the time spend to optimize is included in the run time of the whole query. Currently, the run time consist of the time to translate the query into an abstract syntax tree, the time to translate this AST into a tree of runtime iterators and the time to execute the latter. Optimization takes place on the AST resulting in an AST which can hopefully be translated into a tree of runtime iterators which in turn can be executed more efficiently. This also means that the time used for optimizations should not exceed the time gained due to faster execution. With respect to this, some optimizations may be not suitable. For example constant folding, i.e. replacing for instance `1 + 2` with `3`, may in the end just move the computation from the runtime iterators to the compiler without any performance gain in overall query execution. This experiments and thoughts are left for further work.
Another problem is the environment used. Rumble is written in Java and MLIR is written in C++. To integrate the optimizer in Rumble, one needs to call C++ code from Java. This may lead to the loss of the platform independence provided by Java. Looking further into this is also left for the future.

References

[1] Apache Spark Website. https://spark-apache.org/. Accessed: 2020-12-08.

[2] JSON Website. https://www.json.org/json-en.html. Accessed: 2020-12-08.

[3] JSONiq Website. https://www.jsoniq.org/. Accessed: 2020-12-08.

[4] MLIR Website. https://mlir.llvm.org/. Accessed: 2020-12-08.

[5] Rumble, an engine to run JSONiq on top of Spark. https://blog.systems.ethz.ch/blog/2019/rumble.html. Accessed: 2020-12-08.

[6] Rumble Git Repository. https://gitlab.inf.ethz.ch/gfourny/rumble/-/tree/MLIR3. Accessed: 2020-12-08.

[7] Rumble Website. https://rumbledb.org/. Accessed: 2020-12-08.

[8] TableGen Overview. https://llvm.org/docs/TableGen. Accessed: 2020-12-09.

[9] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko. Mlir: A compiler infrastructure for the end of moore's law, 2020.

[10] I. Müller, G. Fourny, S. Irimescu, C. B. Cikis, and G. Alonso. Rumble: Data independence for large messy data sets, 2020.

[11] M. I. Reber. Optimizing JSONiq Execution in Rumble using MLIR, 2020.

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.