Rhyme: A Data-Centric Expressive Query Language for Nested Data Structures

Supun Abeysinghe[0000−0001−6054−2432] and Tiark Rompf[0000−0002−2068−3238]

Purdue University, West Lafayette IN 47906, USA
{tabeysin,tiark}@purdue.com

Abstract. We present Rhyme, an expressive language designed for high-level data manipulation, with a primary focus on querying and transforming nested structures such as JSON and tensors, while yielding nested structures as output. Rhyme draws inspiration from a diverse range of declarative languages, including Datalog, JQ, JSONiq, Einstein summation (Einsum), GraphQL, and more recent functional logic programming languages like Verse. It has a syntax that closely resembles existing object notation, is compositional, and has the ability to perform query optimization and code generation through the construction of an intermediate representation (IR). Our IR comprises loop-free and branch-free code with program structure implicitly captured via dependencies. To demonstrate Rhyme's versatility, we implement Rhyme in JavaScript (as an embedded DSL) and illustrate its application across various domains, showcasing its ability to express common data manipulation queries, tensor expressions (à la Einsum), and more.

Keywords: Declarative query languages · Logic programming · Tensor expressions · Multi-paradigm languages · Rhyme.

1 Introduction

Declarative programming represents a paradigm in which users articulate *what* computation needs to be performed, without the explicit specification of the procedural steps required for its execution. Declarative programming languages find application across a diverse array of domains. Notable examples include SQL, employed for data querying and manipulation, Datalog [14], used for data querying as well as in domains like declarative program analysis [21,28,34] and binary decompilation [16], Einstein notation (or similar domain specific languages [35]) for expressing tensor computations mathematically, and GraphQL [19] for data querying within the context of web application front-ends, and so on.

In practical scenarios where diverse paradigms of workloads are combined (e.g., data frames + tensors), the necessity arises to employ multiple query languages or systems in tandem. Each of these languages interfaces with the respective engines tasked with handling individual workloads. However, this approach is inherently inefficient, both from a performance perspective, due to the reliance on multiple isolated backends, and from a programmer productivity

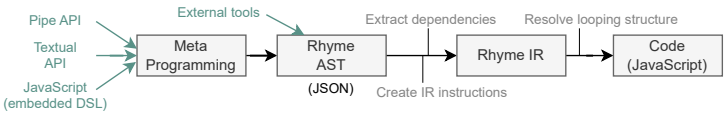


Fig. 1: End-to-end workflow of Rhyme, with green markers indicating various entry points leading to the common entry point, Rhyme AST. The Rhyme AST can be constructed directly from external tools or via metaprogramming using different APIs. This AST serves as the basis for generating an IR (with dependencies), driving subsequent code generation.

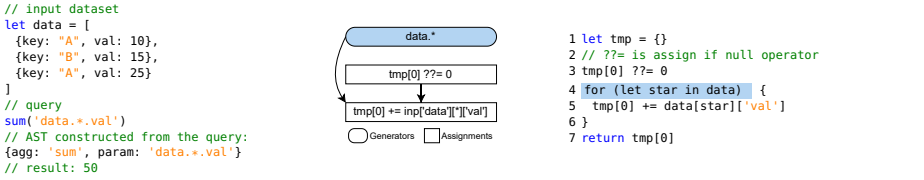
standpoint, as it necessitates learning and maintaining code written in multiple query languages.

A promising approach to address the performance challenge is to construct common intermediate layers across multiple domain-specific systems [8, 27, 32]. In such cases, the respective query languages or system interfaces for different domains act as frontends that emit fragments of intermediate representation (IR) logic corresponding to their portion of the computation. These IR fragments are then assembled to create a unified IR for the combined workload, allowing for global optimizations and the generation of a unified executable. However, a common limitation in these systems is the requirement for individual system developers to implement *all* their operators using a limited set of generic IR operators, which becomes a bottleneck in practice [8].

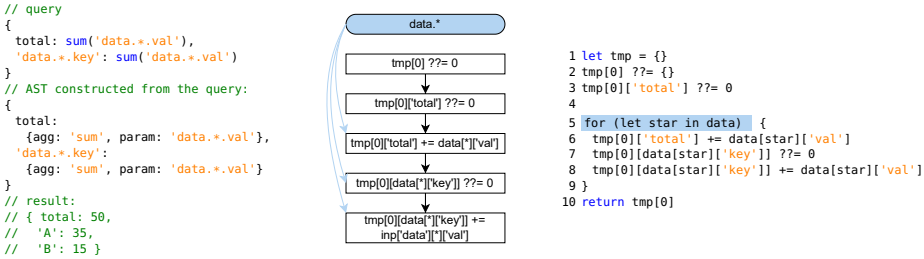
In this work, we address this challenge by introducing a unified query language, named Rhyme [1], that comprises a general substrate capable of accommodating a wide array of different use cases. While our approach shares some similarities with the idea of constructing common IRs, it distinguishes itself by focusing on a higher-level, more expressive common substrate. This distinction allows for the intuitive expression of various workload paradigms, as elaborated throughout the paper. Rhyme takes inspiration from many existing declarative languages including GraphQL, JQ [3], XQuery [4], JSONPath [18], Einstein notation, Datalog, recent functional logic programming languages like Verse [10], etc. Rhyme is designed to serve as an expressive language for high-level data manipulation, enabling the querying and transformation of nested data structures (e.g., JSON, tensors) and producing nested structures as output.

There are several key defining characteristics of Rhyme. First, Rhyme adopts a query syntax that closely mirrors existing object notation, meaning that queries are essentially expressed as JSON objects. Second, Rhyme is designed in a way that permits query optimization and code generation via the construction of an intermediate representation (IR). This IR contains loop-free and branch-free code with dependencies that implicitly capture the program structure. Third, Rhyme is compositional and easy to meta-program, recognizing that data transformation queries are typically used as part of larger programs and are often generated programmatically.

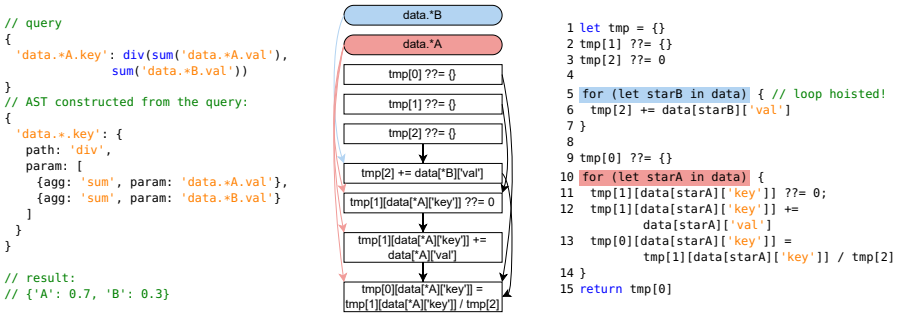
[1] `https://rhyme-lang.github.io`



(a) A query computing the sum of all values



(b) A query computing sum of all values (`total`) and sum per each key



(c) A query computing key-specific relative aggregate proportions

Fig. 2: The end-to-end workflow of Rhyme for three queries, starting with the query and the corresponding AST (left; Section 2.5), followed by the construction of the IR (center; Section 4.1), and the final generated code (right).

Figure 1 provides an overview of the end-to-end workflow of Rhyme. The central point of entry into this workflow is the Rhyme AST (Section 2.5). Notably, this AST is represented in JSON format and is hence serializable, enabling the ability to be exported/imported from various other environments. We implement Rhyme as an embedded DSL in JavaScript (JS), which constructs the Rhyme AST from Rhyme queries. Additionally, alternative interfaces can be used, such as pipes (Section 3.1), or entirely textual inputs that can be processed by a parser to construct the same AST. Once the AST is constructed, it gets transformed into Rhyme IR. During this transformation, declarative query operators are mapped to IR instructions along with their dependencies. Subsequently, this IR is used to analyze the looping structure and generate the final code.

Consider the query in Figure 2a. In Rhyme, `data.*.val` is called a *path* expression, a notation inspired by JSONPath [18]. The $*$ symbol serves as an iterator, facilitating iteration through the `val` values, while the aggregator `sum` calculates the sum of these iterated values. Rhyme's IR (shown in the center) consists of two main types of operators. First, it has *generators* (represented as *rounded* rectangles), which represents iterators that enumerate a list of items (ultimately translated into loops in the generated code). In our example, `data.*` is a generator, iterating values from the data object. Second, the IR has *assignments* (represented as rectangles), comprising the computations required for executing the query. In the case of our first query, the initialization logic `tmp[0] ??= 0` and the subsequent sum computation `tmp[0] += ...` are assignments.

Notably, the IR operates without the need for explicit control flow constructs. Instead, the program's structure is implicitly inferred through dependencies. For our example query, the assignment `tmp[0] += ...` has dependencies to both the initializer (because initialization must come first) and the generator (because its operand is the iterated value). Additionally, it is worth noting that computations are performed on temporary state variables (`tmp[0]` in the first example). This approach, utilizing intermediate temporaries, draws inspiration from works such as RPAI [7] and DBToaster [22]. These systems utilize such state variables to maintain values for various sub-queries of the main query, which are then used to compute the final result. Finally, the IR is translated into JS code (and eventually run using `eval()`), taking dependencies into account to extract the program structure and performing optimizations as part of this transformation process.

Figure 2b and Figure 2c illustrate two additional queries executed on the same dataset. Specifically, in Figure 2b, we compute both the total sum of all values (similar to the query in Figure 2a) and the sum of values per key, effectively a group-by sum operation. A key characteristic of Rhyme lies in the contextual interpretation of expressions such as `sum(data.*.val)`. That is, the semantics of this expression differs depending on its context. This is similar to local unification semantics in Verse [10]. For instance, when it is nested within `{data.*.key: ...}`, the expression signifies a group-by sum operation. Conversely, if it is not nested within an iterator (i.e., $*$), it calculates the aggregate over the entire set of values. This distinction is exemplified in the query presented in Figure 2c. Here, `sum(data.*A.val)` is nested within `{data.*A.key: ...}`, signifying a group-by aggregate operation. Conversely, `sum(data.*B.val)` computes a total aggregate, as it is not nested within a `*B` iterator.

Additionally, Figure 2c also demonstrates an optimization that occurs at the IR level. Specifically, even though the `sum(data.*B.val)` appears as a nested subquery in the original query, the Rhyme backend can determine that the generated `*B` can be hoisted out as a separate loop by analyzing the IR dependencies. This is essentially similar to sub-query hoisting that happens at the logical plan level in other traditional query optimizers.

These examples provide a broad overview of Rhyme's functionalities and its syntactic structure. In the subsequent sections of this paper, we will delve deeper into the syntax and capabilities of Rhyme and discuss the process of IR con-

struction and code generation. Moreover, the previous example queries focused on Rhyme's capability to express analytical queries on JSON objects. However, Rhyme covers an even broader spectrum of use cases, including the ability to express tensor computations and declaratively specify visual components (e.g., tables, charts) of web applications.

Our specific contributions are as follows.

- We introduce the syntax of Rhyme, showcasing the ability to express common data manipulation operators such as selections, group-bys, joins, user-defined functions (UDFs), and others (Section 2).
- We highlight the versatility of Rhyme across various use cases, including the expression of visual elements in web applications (e.g., tables, charts using SVG), declarative tensor computations (akin to Einsum), and alternative 'pipe' APIs via metaprogramming (Section 3).
- We elucidate the process of lowering queries into an IR that features loop-free and branch-free code, with dependencies implicitly representing the program structure. Then, we illustrate how this IR facilitates code generation by constructing the optimal program structure from dependencies (Section 4).
- We evaluate the performance of Rhyme on several JSON analytics workloads to demonstrate the effectiveness of our code generation approach (Section 5).

We discuss related work in Section 6, followed by conclusions and potential future research directions in Section 7.

2 The Rhyme Query Language

In the previous section, we saw Rhyme in action for a set of relatively simple queries. In this section, we will introduce the syntax of Rhyme, illustrating how it facilitates the expression of common data manipulation operations like selections, aggregates, group-bys, and so on. The formal grammar is shown in Figure 3. The rest of this section illustrates how each of these components are used to express different kinds of queries. To improve understanding, we will employ a running illustrative example dataset, as depicted below. The dataset contains populations of several major cities, along with the respective country. Our chosen dataset is deliberately kept simple, devoid of intricate nested structures. However, Rhyme has the capacity to seamlessly query nested JSON data in the same way. We will see such examples later in Section 3.

```
let data = [
  {country: "Japan",  city: "Tokyo",      population: 14},
  {country: "China",  city: "Beijing",    population: 22},
  {country: "France", city: "Paris",      population: 3},
  {country: "UK",     city: "London",     population: 9},
  {country: "Japan",  city: "Osaka",      population: 3},
  {country: "UK",     city: "Birmingham", population: 2}
]
```

```
Ident ::= [a-zA-Z_][a-zA-Z0-9_]*
Num ::= [0-9]+
Var ::= * Ident
ScalarOp ::= get | apply | plus | minus
           | div | fdiv | times | mod
ReductionOp ::= sum | count | max | min
              | first | last | array

Atom ::= Ident | Num | Var
Path ::= Atom ( . Atom )*
Expr ::= Path
       | Expr ScalarOp Expr
       | ReductionOp ( Expr )
       | [ Expr ]
       | { ( Path : Expr )* }
Query ::= Expr
```

Fig. 3: Syntax for expressing Rhyme queries.

2.1 Basics

First we will look at how to perform several basic query operations on the aforementioned dataset. For instance, if we want to select a particular key of the dataset at a given index, we can use the following syntax.

```
'data.2.country'              // result: France
{first : 'data.0.country'}    // result: {first: Japan}
```

Here, the reference `data` refers to the dataset object, and can be simply indexed through integer indices. Furthermore, specific keys can be selected by specifying the desired key names (e.g., `.country`). Notably, Rhyme offers the convenience of the familiar JS-like syntax for constructing structured output from extracted values, as exemplified in the second instance.

While this form of explicit indexing into the array can be useful for several use cases, generally, queries involve some form of iterating over the dataset. Rhyme offers this capability through the $*$ operator, serving as an implicit iteration operator. Moreover, as we saw in Figure 2c, we can perform controlled iteration with multiple generator symbols (e.g., `*A`, `*B`, etc.). In fact, these generator symbols behave like logic variables in Datalog, Prolog, and other logic programming languages as we see in Section 3.2. Below, we present three example queries that leverage iterators and compute aggregates over the iterated values.

```
['data.*.city']          // result: [Tokyo, Beijing, ..., Birmingham]
sum('data.*.population') // result: 53
max('data.*.population') // result: 22
```

These queries are self-explanatory in nature. In the first example, we illustrate a scenario where an array can be constructed from the values obtained through iteration, employing the `[...]` syntax. Moreover, users can compute aggregates over the iterated values using the relevant aggregate functions, such as `sum`, `max`, and so forth. As discussed previously, these queries can be used as parts of object construction logic and combined flexibly, as shown below.

```
{ total: sum('data.*.population'),          // result:
  highest: max('data.*.population') }       // {total: 53, highest: 22}
```

2.2 Group By

Another vital query operator, especially relevant to JSON-style objects, is the group-by query. Rhyme offers an intuitive means of implicitly expressing group-

bys. The following query exemplifies this, grouping records based on the `country` attribute and subsequently calculating the total population for each group:

```
{ 'data.*.country': sum('data.*.population') }
// result: {Japan: 17, China: 22, France: 3, UK: 11}
```

Here, specifying `{data.*.country: ...}` as the key implies that any iteration carried out within this key utilizing the same iterator (∗) is performed for records with each unique value of `country` separately. The next example shows how to use this form of grouping to compute aggregates at different levels. It computes the total population of all records, breaks it down by country, and subsequently computes the population proportion of each city with respect to total population:

```
let query = {
 total: sum('data.*.population'),                  // total population
 'data.*.country': {
   total: sum('data.*.population'),                // population per country
   'data.*.city': div(sum('data.*.population'),    // population proportion
                      sum('data.*A.population'))}  // (per each city)
}
// result: {total: 53, Japan: {total: 17, Tokyo: 0.26, Osaka: 0.06}, ...}
```

In the given query, the `sum('data.*.population')` at each query level computes distinct results: total population, total population per country, and population per city, respectively. If we had multiple population values for a given city (e.g., county data), then the last aggregation would compute the per city sum. As previously discussed, since `sum('data.*A.population')` is not nested within a `*A` key, it performs a total aggregation, which sums all the population values. It is worth noting that our implementation employs calls like `div` for some operators since it functions as an embedded DSL in JS. In a textual frontend, the query could appear even more concise, using standard operators like `/` for division.

2.3 Join

Joins are another fundamental operator in data querying. To illustrate how joins work in Rhyme, consider the following new dataset named `other`, which includes information about the `region` to which each country belongs:

```
let other = [
 {country: "Japan",  region: "Asia"},
 {country: "China",  region: "Asia"},
 {country: "France", region: "Europe"},
 {country: "UK",     region: "Europe"},
]
```

Now, consider a scenario where we aim to compute aggregate population values based on regions. For this, we must perform a join between our original `data` and this new `other` object to acquire the corresponding region for each country. The following Rhyme query illustrates how this is expressed.

```
let countryToRegion = {                       // create a mapping of
 'other.*O.country': 'other.*O.region'        // country -> region
}
let query = {
 '-': keyval(get(countryToRegion, 'data.*.country'), { // Use of "-" and keyval is because
   total: sum('data.*.population')                     // JS enforces JSON keys to be
    'data.*.country' : sum('data.*.population')        // strings and our key is a var
   })
}
// result: {Asia: {total:39, Japan:17, China:22}, Europe: {total:14, France:3, UK:11}}
```

Here, we use a distinct query (`countryToRegion`) to retrieve the corresponding region for a given country. The main query conducts a group-by based on the `region` (retrieved using `get`) first, followed by another group-by based on `country`, ultimately computing the desired aggregates. We use `'-'` in our implementation due to JS's requirement that JSON keys be represented as strings. Consequently, specifying `get(countryToRegion, 'data.*.country')` as the key directly is not possible. Hence, we introduce `keyval(<key>, <value>)` as a workaround that allows arbitrary arguments to be used as a key. It is worth noting that in a textual frontend, such workarounds would not be necessary.

2.4 User-defined Functions

Rhyme allows using user-defined functions (UDFs) written in JS seamlessly with the queries. Consider a simple query where we want to obtain the percentage population per each country from our dataset.

```
let udf = {
  formatPercent: v => (v*100).toFixed(2) + "%" // computes the percentage
}
let query = {
  'data.*.country':
    apply(udf.formatPercent, div(sum('data.*.population'), sum('data.*A.population')))
}
// result: {Japan: 32.05%, China: 41.50%, France: 5.66%, UK: 20.75%}
```

We have defined the UDF `formatPercent` that, given a proportion value, computes the percentage and adds a % sign at the end. We can then use `apply` in the query to call this UDF to convert the proportions to percentages.

2.5 Rhyme AST

As we saw in Figures 1 and 2, all the queries above construct a Rhyme AST representation which serves as the basis for the dependency analysis and IR construction. This AST representation is in JSON format, and closely mirrors the query JSON structure. The difference is, all the calls to reducers like `sum`, `count`, etc. and other operations like `plus`, `minus`, etc. will be translated to explicit object components with `agg` (or `path`) and the corresponding arguments. Here, `agg` is for reducers (which are stateful), and `path` is for operations simply performs a lookup and some computation. For instance, Figure 2c (left) shows how `sum` is translated to `agg` and `param`, and `div` is translated to `path` and `param`.

This AST representation serves as the entry point for our compiler backend, and gets translated into an IR, as elucidated in Section 4. Moreover, as depicted in Figure 1, the creation of this AST is not limited to the aforementioned JS embedding. Instead, it can be constructed using different APIs, (e.g., Fluent API introduced in Section 3.1), or a textual frontend with a parser, and so on.

3 Case Studies

3.1 Fluent API

Rhyme's query frontend (JSON) we saw in Section 2 describes the query using the structure of the computed *result*. However, sometimes, it is more natural to

start from the structure of the *input*, and specify a sequence of transformation steps. Given that our frontend is embedded in JS, we can use metaprogramming to layer a LINQ-style [24] pipeline API on top.

To illustrate the advantages of such an interface, consider a simple task borrowed from Advent of Code 2022 [1]. The task involves processing a sequence of values partitioned into chunks, each containing multiple values. The objective is to calculate the sum of values for each chunk and subsequently identify the maximum sum among those computed. To begin, let us examine how the Rhyme query appears when utilizing the familiar JSON-style API for data parsing and computation. First, we define several user-defined functions (UDFs) to assist with data parsing. The role of each UDF is simple and self-explanatory.

```
let input = '100,200,300|400|500,600|700,800,900|1000' // sample input
// some UDFs for parsing the data
let udf = {
  'splitPipe' : x => x.split('|'),
  'splitComma': x => x.split(','),
  'toNum'     : x => Number(x)
}
```

Shown below is the Rhyme query responsible for executing the required computation, with comments provided alongside to elucidate each section of the query (numbered for clarity):

```
1 let query = max(get({           // 5. find maximum among group sums
2  '*chunk': sum(                 // 4. group-by chunk and compute sum
3    apply('udf.toNum',           // 3. convert each string number to a number object
4     get(apply('udf.splitComma', // 2. split by comma to get numbers of each chunk
5       get(apply('udf.splitPipe', '.input'), '*chunk')), // 1. split into chunks
6       '*line')))
7  },'*'))
```

Function `apply()` is used to apply UDFs to arguments; `get()`, when used with an unbounded generator symbol (e.g., `*chunk`), binds the iterator to the object in the first argument. For example, in Line 5, `get(..., '*chunk')` binds the iterator `*chunk` to the result of splitting the output by the pipe symbol (|). While this approach works as intended and yields the correct results, for these kinds of workloads, it is more natural to think starting from the input instead of the output structure. In such cases, Rhyme's 'fluent' interface offers an alternative way to express this query concisely as shown below. Here, we specify the query as a sequence of transformation steps on the input. This high-level fluent API essentially functions as a metaprogramming layer that generates an equivalent Rhyme AST as before.

```
let query =
  pipe('.input')
    .map('udf.splitPipe').get('*chunk') // 1. split into chunks (and bind to *chunk)
    .map('udf.splitComma').get('*line') // 2. split by comma to get numbers of each chunk
    .map('udf.toNum')                   // 3. convert each string number to a number object
    .sum().group('*chunk').get('*')     // 4. group-by chunk and compute sum
    .max()                              // 5. find maximum among group sums
```

The `pipe()` function creates a `Pipe` object equipped with methods `sum`, `max`, and so on, all of which return a `Pipe`. The `map()` function, similar to `apply()` mentioned earlier, is employed to apply a UDF, and `group()` is used to perform a group-by (i.e., `e.group(x)` is `{x : e}`).

3.2 Tensor Expressions

Rhyme provides an elegant framework for expressing tensor computations, drawing inspiration from the Einstein summation (Einsum) notation frequently employed in tensor frameworks and Einops [29]. The Einsum notation offers a concise means of articulating tensor computations. For instance, $ik, kj \rightarrow ij$ specifies a standard matrix product that takes two two-dimensional tensors (i.e., matrices A and B), and yields a third tensor (say C), as the result, computed as $C_{ij} = \sum_k A_{ik} \times B_{kj}$. Likewise, complex tensor computations involving multiple n-dimensional tensors can be specified using such declarative expressions.

Rhyme provides a similar way to express tensor computations in a declarative fashion. To perform tensor computations, we rely on using the notion of unbounded iterators in Rhyme. Specifically, in prior cases, we explicitly specified the data source from which we iterate, as seen in constructs like `data.*A`. However, if instead we only specify the iterator as the key, Rhyme's backend automatically determines the appropriate data source by examining the query body. For instance, when we have a query like `{*i: sum(times(A.*i, B.*i))}`, the backend selects either `A` or `B` as the data source for iteration. Subsequently, the generated code ensures that the iterated values exist in both `A` and `B`. This concept essentially parallels the notion of unification in logic programming, and more specifically narrowing in functional logic programming [10, 20].

To demonstrate how tensor computations are expressed in Rhyme, let's consider some examples. While Rhyme accommodates tensors in various nested formats, for the sake of simplicity, we will consider a scenario where we represent tensors using JS Arrays, as illustrated below:

```
let A = [[1, 2], [3, 4]]; let B = [[1, 2, 3], [4, 5, 6]]
```

Shown below are a set of example tensor computations expressed in Rhyme. Einsum notation counterparts (which closely mirror Rhyme query structure) are shown in parentheses for each example.

```
// tensor transpose (ij->ji)
{'*j': {'*i': 'B.*i.*j'}}
// sum of all elements (ij->)
sum('B.*i.*j')
// column sum (ij->j)
{'*j': sum('B.*i.*j')}
// row sum (ij->i)
{'*i': sum('B.*i.*j')}
// dot product (vector-vector) (i,i->)
sum(times('vecA.*i', 'vecB.*i'))
```

```
// matrix multiplication (ik,kj->ij)
{'*i': {'*j': sum(
  times('A.*i.*k', 'B.*k.*j')) }}
// Hadamard product (ij,ij->ij)
{'*i': {'*j': times('A.*i.*j', 'B.*i.*j') }}
// general tensor contraction (n-d Tensors)
// e.g., pqrs,tuqvr->pstuv
{'*p':{'*s':{'*t':{'*u':{'*v':sum(
  times('T1.*p.*q.*r.*s',
        'T2.*t.*u.*q.*v.*r')) }}}}}
```

One benefit of having a unified query language for both data manipulation and tensor computations is the ability to handle combined workloads efficiently. To illustrate this, consider a simplified version of computing a city's 'crime index' (taken from [27]). We first select a set of features of cities (e.g., population, adult population and number of robberies), followed by a dot product with a predefined weight vector.

```
let cityVec = { 'data.*.city': ['data.*.pop', 'data.*.adultPop', 'data.*.numRobs'] }
let weightVec = [1.0, 1.0, -2000.0] // weight of each feature
let crimeIndex = { // dot product
  'data.*.city': sum(times('weightVec.*i', get(get(cityVec, 'data.*.city'), '*i')))
} // computes the crime index for each city
```

We can specify both the data manipulation component (i.e., the projection) and the tensor computation (i.e., dot product) within the same query language, and we generate a unified code for the combined task. While we kept the example simple for brevity, this has the potential to optimize practical intricate workloads that combine data processing with tensor computations.

3.3 Declarative Visualizations

Since Rhyme is embedded within JS, we can extend its capabilities by introducing a means to *declaratively* specify the visual components of websites using a similar structural approach. This allows the seamless integration of data querying logic with the corresponding data visualization logic, such as creating tables. This is enabled by a special key called `'$display'`. To illustrate this, consider the following example query, alongside its corresponding output:

```
{
  '$display': 'table'   // display data in a table
  rows: [0], cols: [1], // row:data index, col:keys
  data: [
    {region:"Asia",city:"Beijing",
      "population":{'$display':"bar",value:40}},
    {region:"Asia",city:"Tokyo",
      "population":{'$display':"bar",value:70}}
  ]
}
```

	region	city	population
0	Asia	Beijing	
1	Asia	Tokyo	

The query above produces the visualization shown on the right. Specifically, it declaratively specifies to display a table that has `data` as the underlying data. This data can be some raw JSON or another Rhyme query. Notice that we can mix, and manipulate these components as valid values inside Rhyme queries, and compose them as necessary. For instance, the progress bars are manipulated as values in the query. These visualizations can take various forms, including standard DOM elements like `h1`, `p`, or high-level components like `table`, `bar`, `select`, and more, as well as SVG objects.

To exemplify the practical utility of this approach, consider a scenario involving the visualization of data related to mobile phone supplier warehouses. Imagine the raw data is represented as an array of JSON objects, each featuring attributes such as warehouse, product, model, and quantity. Before delving into the main query, shown below is a helper query. This auxiliary query calculates the sum of quantities, generates a formatted percentage total, and presents this information as a progress bar displaying the corresponding percentage.

```
let computeEntry = {
  'Quantity': sum('data.*.quantity'),
  'Percent': apply('udf.formatPercent',
       div(sum('data.*.quantity'),sum('data.*B.quantity'))),
  'Bar Chart': {
    '$display': 'bar',
    value: apply('udf.percent', div(sum('data.*.quantity'), sum('data.*B.quantity')))
  }
}
```

As previously discussed, the semantics of these aggregations varies depending on the calling context. For instance, when invoked within the context of `{'data.*.warehouse': ...}`, the aggregates computed on $*$ iterators are grouped

based on the `warehouse` attribute. Below, we present a query that leverages the above sub query and visualizes our data in a pivot table. This query performs aggregations at multiple levels and displays each level of aggregate in a single table as shown on the right. Naturally, this repeated structure could be abstracted further into a single operation such as `rollup('data.*', [model, product, warehouse], computeEntry)`.

```
let query = {
  '$display': 'table',
  ...
  data: { Total: {
    props: computeEntry, // total aggregate
    children: {'data.*.warehouse': {
      props: computeEntry, // warehouse-level aggr
      children: 'data.*.product': {
        props: computeEntry, // product-level aggr
        children: {
          'data.*.model': // model-level aggr
             computeEntry
        }
  ...
}
```

	Quantity	Bar Chart	Percent Total
Total	1210		100 %
San Jose	650		53 %
iPhone	300		24 %
7	50		4 %
6s	100		8 %
X	150		12 %
Samsung	350		28 %
Galaxy S	200		16 %
Note 8	150		12 %
San Francisco	560		46 %
iPhone	260		21 %
7	10		0 %
6s	50		4 %

We can similarly use Rhyme to visualize data in different types of charts using SVG graphics (such examples are omitted due to space concerns). Regarding how this visualizations are handled in the backend, we start by building the necessary helper functions to create these components (tables, bars, etc.) programmatically. Then, the backend is augmented to use these functions whenever a `'$display'` is encountered. Our ultimate goal of these kinds of integrations is to enable users to use Rhyme to build interactive dashboards and CRUD applications directly from a single query.

4 IR and Code Generation

Up to this point, we have provided an introduction to the syntax of Rhyme and explored its versatility across various domains. In Figure 1 and Figure 2, we gained a preliminary understanding of how Rhyme queries are transformed into an IR, which subsequently serves as the basis for generating optimized code. In this section, we will delve into a detailed discussion of the IR structure and the code generation process.

4.1 IR Structure

As discussed in Section 1 (Figure 2), the IR structure of Rhyme consists of two primary types of instructions: *generators* and *assignments*. Generators correspond to iterators responsible for enumerating input or intermediate nested objects, and these generators are transformed into loops in the generated code. Assignments, on the other hand, encompass any form of computation that updates or initializes an intermediate or output state.

Rhyme queries inherently exhibit nested iterating structures that could be simply translated into a series of nested loop structures in the generated code. However, performing this transformation naively and enforcing the 'program

structure' implied by the user query would lead to missed optimization opportunities like hoisting computations and loops that are independent of outer loops, common sub-query/expression elimination, and more. Therefore, rather than naively transforming queries and directly imposing the program structure, we extract a set of generators, iterators, and their dependencies during IR construction. While the IR does not explicitly capture the program structure, the optimal program structure can be derived from an analysis of these dependencies.

4.2 Constructing the IR

The dependency structure is relatively straightforward. As demonstrated in the generated code snippet in Figure 2, we utilize objects such as `tmp[0]`, `tmp[1]`, and so on, to maintain intermediate results required for computing the final query result. Assignment operators have these temporaries as operands, creating data dependencies in the process. Generators can also iterate values from these temporaries, and in such cases, we introduce similar dependencies for the generators. Similarly, when a generator symbol is used in an assignment or another generator, a dependency is added. To illustrate how dependencies are created, consider the following assignment instruction extracted from Line 13 in Figure 2c:

```
13 tmp[0][data[starA]['key']] = tmp[1][data[starA]['key']] / tmp[2]
```

This instruction relies on `tmp[0]`, `tmp[1]`, and `tmp[2]` as operands. As a result, this instruction is associated with the last (write) operations of all three temporaries as dependencies, as depicted in the IR visualization presented in Figure 2c. Furthermore, since it employs the `*A` iterator, it also exhibits a dependency on the corresponding generator.

The final missing piece in our lowering process is determining the appropriate IR instruction from our query AST. This is done distinctly for reduction operators (those with a `key` in the AST) and other operators (those with a `path` in the AST). For reduction operators, which require the management of state, we introduce stateful temporary variables indexed by the current grouping path. In this context, 'path' refers to the pertinent 'parent' grouping keys within the query nest. In contrast, other types of operators are simpler to handle. We retrieve the operands and subsequently create an instruction that performs the desired computation. For example, for `plus`, we retrieve the left and right operands and create a binary operation utilizing the `+` operator.

4.3 Code Generation

Once the IR is constructed for a query, the next step is to generate the final code taking into account the instruction dependencies. Specifically, it entails determining where to insert loops, how to nest loops within one another, where to place assignment instructions, and so on. We will use the query from Figure 2c as a running example for this section.

The first step is computing two auxiliary relations: `tmpInsideLoop` and `tmpAfterTmp`. These relations track which temporaries should be scheduled inside par-

ticular loops and which temporaries should be scheduled after certain other temporaries. This can be done by analyzing assignment to assignment dependencies and generator to assignment dependencies.

The next step involves computing the relation `tmpAfterLoop` based on the two relations computed above. Specifically, if we determined that a temp variable `t2` should be scheduled after another temp variable `t1` (i.e., `tmpAfterTmp[t2][t1]`), which resides inside a loop `l` (with the condition that `t2` itself is not within loop `l`), then this implies that `t2` should be scheduled after the loop `l`. For instance, in our sample query, `tmp[0]` should be scheduled after `*A`.

Subsequently, as the final analysis step before code generation, we determine `loopAfterLoop` and `loopInsideLoop`. These essentially help identify how the loops should be scheduled. In particular, if we ascertain that a given temp `t` should be scheduled inside both loops `l1` and `l2`, it implies that `l1` and `l2` should belong to the same loop nest. Conversely, if we determined that for a particular temp `t`, it resides within loop `l2`, and we also know that `t` should be scheduled after another loop `l1`, then this indicates that the loop `l2` should be scheduled after `l1` (provided they are not part of the same loop nest).

Once the analysis steps are completed, we proceed with the code generation process. Our approach to code generation draws inspiration from the IR scheduling algorithm utilized in Lightweight Modular Staging (LMS) [11,12,31]. In particular, we schedule generators and assignments in an 'outside-in' fashion, commencing with the outer loops before progressing to the inner ones.

Since we did not have program control structures enforced from the front end, this code scheduling mechanism freely schedules assignments and generators in an optimal manner. For instance, any generator that does not have dependencies to the 'outer query' would be hoisted and scheduled as a separate query instead of repeating the computation multiple times inside a nested loop.

5 Experiments

In this section, we conduct a performance evaluation of our current Rhyme implementation, comparing it against two established JSON processing systems: JQ [3] and Rumble [25], the latter of which utilizes Spark for distributed processing. We acknowledge that systems like Rumble are primarily designed for large-scale, cluster execution and may not exhibit optimal performance in a single-node, single-threaded context. Nevertheless, we consider it a valuable baseline for comparison. Moreover, we report the end-to-end execution time, including time for loading data.

5.1 Experimental Setup

We run three queries on a simple synthetic dataset comprising 1 million records of JSON objects. Each object in the dataset contains two string keys, `key1` and `key2`, as well as an integer `value`. The first query calculates the sum of all `value`s,

the second query performs an aggregate sum after grouping by the `key1`, and the third query computes a two-level aggregate using `key1`, then `key2`.

We run all the experiments using a single thread, on a NUMA machine with 4 sockets, 24 Intel(R) Xeon(R) Platinum 8168 cores per socket, and 750GB RAM per socket (3 TB total) running Ubuntu 18.04.4 LTS. We have used JQ v1.6, Rumble v1.21.0, and Node v18.18.0 (for running our JS code). All experiments are run five times, reporting the mean execution time.

5.2 Results

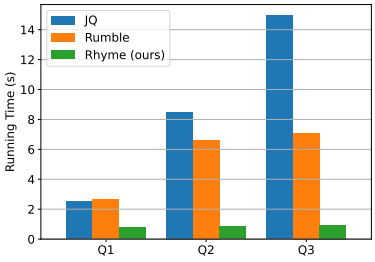


Fig. 4: Running time (left) for JQ, Rumble, and Rhyme for three different queries

Across the three queries, Rhyme demonstrates the best performance. This can be attributed to its ability to generate optimized JS code tailored to a specific query, which significantly reduces overhead compared to the general execution engines employed by systems like Rumble. JQ performs the worst, as it lacks any form of 'query planning' and executes queries naively without optimizations such as loop fusion.

While these results highlight the potential performance gains achievable through Rhyme's code generation capabilities, it is important to note that this benchmark does not provide a comprehensive analysis that contains the full spectrum of representative cases in JSON analytics. Such a comprehensive evaluation is deferred to future research.

6 Related Work

There are several query languages designed for working with semi-structured data like JSON, each with its own focus and strengths. JSONiq [5,17] is a notable query language explicitly tailored for JSON data, borrowing most of its syntax from XQuery [4] (e.g., FLWOR expressions). Zorba [6] and RumbleDB [25] are examples of engines that support JSONiq, with RumbleDB using Spark [37] as a backend, leveraging the scalability of Spark for execution. AsterixDB, designed for semi-structured data, employs AQL [2] and SQL++ [26] as its query languages. GraphQL [19], on the other hand, is widely used in web application development for querying data from backend services. Most of these languages are specifically targeted towards large-scale JSON analytics workloads and are not expressive enough to support cases like the ones in Sections 3.1 to 3.3. While we take inspiration from these languages for the design of the language, Rhyme is designed to handle various forms of nested data (e.g., tensors), and it offers support for efficient code generation and optimizations through its IR.

Rhyme's path expressions are inspired by JSONPath [18] (a descendent of XPath). Although not discussed extensively in this paper, it is possible to extend

Rhyme to support the full set of JSONPath operators, which include conditions, recursion, etc. Rhyme also borrows many ideas from functional logic programming languages like Verse [10], Curry [20], miniKanren [13] and Scalogno [9], and adapts them into a new data-centric declarative language.

7 Conclusions and Future Work

In this paper, we introduced Rhyme, a new query language tailored for high-level data manipulation. We illustrated how Rhyme's design facilitates query optimization and code generation through the construction of an IR. This IR comprises loop-free branch-free code, with program structure implicitly captured by dependencies. Throughout the paper, we demonstrated the versatility of Rhyme by showcasing its applicability in expressive data manipulations, tensor computations, manipulation of visual aspects, and so on in a declarative manner. It is worth noting that Rhyme is still in its early developmental stages, and we are excited about exploring various avenues of interesting future work.

Incrementality While not extensively covered in this paper, the concept of utilizing intermediate temporaries within the generated code is inspired by prior works in incremental execution, such as DBToaster [22] and RPAI [7]. An immediate focus of our future work involves introducing support for incremental execution. Notably, our generated code is inherently designed to be 'incremental-friendly'. This implies that we have the capability to generate code akin to update triggers, which are invoked whenever a modification is made to the dataset. Specifically, instead of dense loops used in the current version, update triggers generate 'sparse' loops in the sense that they iterate only over the deltas.

Another dimension of incrementality involves managing query changes. This entails finding ways to accommodate changes in queries while maximizing the utilization of previously computed temporaries and sharing state across multiple queries. Such an approach will be useful in interactive applications, where users can dynamically modify their queries.

Performance While the ability to generate JS for browser-based execution is undeniably valuable, there are specific scenarios where optimizing performance becomes paramount. In such cases, the generation of low-level, specialized C code becomes imperative to eliminate any potential overhead associated with managed runtimes. A substantial body of prior research has already demonstrated the efficacy of such compilation mechanisms [15,30,33,36]. Furthermore, it is feasible to leverage existing compiler infrastructures such as LMS [31] or MLIR [23] for streamlined handling of tasks like IR construction, dependency analysis, and the eventual generation of highly specialized low-level code.

Acknowledgements We would like to thank our anonymous reviewers for their valuable feedback that helped improve the paper significantly. This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Meta, Google, Microsoft, and VMware.

References

1. Advent of code 2022. https://adventofcode.com/2022/day/1, accessed: 2023-09-27
2. The asterix query language (aql). https://asterixdb.apache.org/docs/0.9.8/aql/manual.html, accessed: 2023-09-27
3. jq manual. https://jqlang.github.io/jq/manual/, accessed: 2023-09-27
4. Xquery 3.1: An xml query language. https://www.w3.org/TR/xquery-31/ (2017), accessed: 2023-09-27
5. Jsoniq. https://www.jsoniq.org/ (2018), accessed: 2023-09-27
6. Zorba. https://www.zorba.io/ (2018), accessed: 2023-09-27
7. Abeysinghe, S., He, Q., Rompf, T.: Efficient incrementialization of correlated nested aggregate queries using relative partial aggregate indexes (RPAI). In: SIGMOD Conference. pp. 136–149. ACM (2022)
8. Abeysinghe, S., Wang, F., Essertel, G.M., Rompf, T.: Architecting intermediate layers for efficient composition of data management and machine learning systems. CoRR **abs/2311.02781** (2023)
9. Amin, N., Byrd, W.E., Rompf, T.: Lightweight functional logic meta-programming. In: APLAS. Lecture Notes in Computer Science, vol. 11893, pp. 225–243. Springer (2019)
10. Augustsson, L., Breitner, J., Claessen, K., Jhala, R., Peyton Jones, S., Shivers, O., Steele Jr., G.L., Sweeney, T.: The verse calculus: A core calculus for deterministic functional logic programming. Proc. ACM Program. Lang. **7**(ICFP) (2023)
11. Bracevac, O., Wei, G., Jia, S., Abeysinghe, S., Jiang, Y., Bao, Y., Rompf, T.: Graph irs for impure higher-order languages (technical report). CoRR **abs/2309.08118** (2023)
12. Bračevac, O., Wei, G., Jia, S., Abeysinghe, S., Jiang, Y., Bao, Y., Rompf, T.: Graph irs for impure higher-order languages: Making aggressive optimizations affordable with precise effect dependencies. Proc. ACM Program. Lang. **7**(OOPSLA2), 236:1–236:31 (2023)
13. Byrd, W.E.: Relational programming in miniKanren: techniques, applications, and implementations. Ph.D. thesis, Indiana University (2009)
14. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about datalog (and never dared to ask). IEEE Trans. Knowl. Data Eng. **1**(1), 146–166 (1989)
15. Essertel, G.M., Tahboub, R.Y., Decker, J.M., Brown, K.J., Olukotun, K., Rompf, T.: Flare: Optimizing apache spark with native compilation for scale-up architectures and medium-size data. In: OSDI. pp. 799–815. USENIX Association (2018)
16. Flores-Montoya, A., Schulte, E.M.: Datalog disassembly. In: USENIX Security Symposium. pp. 1075–1092. USENIX Association (2020)
17. Florescu, D., Fourny, G.: Jsoniq: The history of a query language. IEEE Internet Comput. **17**(5), 86–90 (2013)
18. Goessner, S.: Jsonpath - xpath for json. https://goessner.net/articles/JsonPath/ (2007), accessed: 2023-09-27
19. GraphQL: A query language for your api. https://graphql.org/, accessed: 2023-09-27
20. Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics. Lecture Notes in Computer Science, vol. 7797, pp. 123–168. Springer (2013)
21. Jordan, H., Scholz, B., Subotic, P.: Soufflé: On synthesis of program analyzers. In: CAV (2). Lecture Notes in Computer Science, vol. 9780, pp. 422–430. Springer (2016)

22. Koch, C., Ahmad, Y., Kennedy, O., Nikolic, M., Nötzli, A., Lupei, D., Shaikhha, A.: Dbtoaster: higher-order delta processing for dynamic, frequently fresh views. VLDB J. **23**(2), 253–278 (2014)
23. Lattner, C., Pienaar, J.A., Amini, M., Bondhugula, U., Riddle, R., Cohen, A., Shpeisman, T., Davis, A., Vasilache, N., Zinenko, O.: MLIR: A compiler infrastructure for the end of moore's law. CoRR **abs/2002.11054** (2020)
24. Meijer, E., Beckman, B., Bierman, G.M.: LINQ: reconciling object, relations and XML in the .net framework. In: SIGMOD Conference. p. 706. ACM (2006)
25. Müller, I., Fourny, G., Irimescu, S., Cikis, C.B., Alonso, G.: Rumble: Data independence for large messy data sets. Proc. VLDB Endow. **14**(4), 498–506 (2020)
26. Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ semi-structured data model and query language: A capabilities survey of sql-on-hadoop, nosql and newsql databases. CoRR **abs/1405.3631** (2014)
27. Palkar, S., Thomas, J., Shanbhag, A., Narayanan, D., Pirk, H., Schwarzkopf, M., Amarasinghe, S.P., Zaharia, M.: A common runtime for high performance data analysis. In: CIDR. www.cidrdb.org (2017)
28. Reps, T.W.: Solving demand versions of interprocedural analysis problems. In: CC. Lecture Notes in Computer Science, vol. 786, pp. 389–403. Springer (1994)
29. Rogozhnikov, A.: Einops: Clear and reliable tensor manipulations with einstein-like notation. In: ICLR. OpenReview.net (2022)
30. Rompf, T., Amin, N.: A SQL to C compiler in 500 lines of code. J. Funct. Program. **29**, e9 (2019)
31. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. Commun. ACM **55**(6), 121–130 (2012)
32. Sujeeth, A.K., Brown, K.J., Lee, H., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: Delite: A compiler architecture for performance-oriented embedded domain-specific languages. ACM Trans. Embed. Comput. Syst. **13**(4s), 134:1–134:25 (2014)
33. Tahboub, R.Y., Essertel, G.M., Rompf, T.: How to architect a query compiler, revisited. In: SIGMOD Conference. pp. 307–322. ACM (2018)
34. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, Volume II. Computer Science Press (1989)
35. Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W.S., Verdoolaege, S., Adams, A., Cohen, A.: Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. CoRR **abs/1802.04730** (2018)
36. Wei, G., Chen, Y., Rompf, T.: Staged abstract interpreters: fast and modular whole-program analysis via meta-programming. Proc. ACM Program. Lang. **3**(OOPSLA), 126:1–126:32 (2019)
37. Zaharia, M., Xin, R.S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M.J., Ghodsi, A., Gonzalez, J., Shenker, S., Stoica, I.: Apache spark: a unified engine for big data processing. Commun. ACM **59**(11), 56–65 (2016)