JAL: an Algebra for JSON Query Optimization

Anne Jasmijn Langerak[1], Flavius Frasincar[1*], Jasmijn Klinkhamer[1]

[1*]Department of Econometrics, Erasmus University Rotterdam,
Burgemeester Oudlaan 50, Rotterdam, 3062 PA, the Netherlands.

*Corresponding author(s). E-mail(s): frasincar@ese.eur.nl;
Contributing authors: annejasmijn@vanlangerak.com; 584059jk@eur.nl;

Abstract

As databases become larger and less structured, the JavaScript Object Notation (JSON) data format has risen in usage compared to other data formats like XML. At the same time, while extracting data from these large datasets efficiently is of obvious importance, there has been far less research regarding the optimization of JSON queries than there has relating to the querying of XML data. Thus a JSON Data Model and JSON Algebra (JAL) are proposed, as well as a heuristic optimization algorithm, for the purpose of improving the efficiency of queries of JSON data. We implement the proposed algorithm and compare the efficiency gain that it provides in terms of both the theoretical and physical cost of executing queries. We find that the algorithm significantly reduces query costs compared to an unoptimized baseline. Additionally, we find that the efficiency gain is considerably larger when querying databases with many documents than those with relatively fewer documents.

Keywords: JSON, query optimization, JSONiq, databases

1 Introduction

As Big Data increasingly finds its way into the practices of companies across a widening spectrum of industries, the interest in efficient data processing has increased substantially [1]. Of particular importance is the extraction of desired information from large datasets, where the use of queries can be especially costly when not done efficiently.

At present, there are two data transfer types in common usage, XML [2] and JavaScript Object Notation (JSON) [3]. They share many similarities, as JSON was proposed after XML. However, in recent years JSON has become more widespread

compared to XML, a situation heightened by the increase in usage of Representational State Transfer (REST) APIs. Such APIs depend on easy and fast data interchanges, and as JSON is a lightweight, easy to read, and easy to parse data format, it is particularly suited to these APIs. Although JSON is becoming increasingly more popular than XML, current literature still lacks the same depth of research as is available for XML. XML is known for having powerful validation and schema features, with an established set of query languages and resources. The querying of JSON is comparatively less consolidated, and there is still room for efficiency gain regarding the execution of JSON queries.

The focus of this paper is the optimization of JSON queries through the algebraic manipulation of queries, i.e., the logical optimization of queries. To that end, a JSON data model, algebraic operators, equivalence rules, and a heuristic optimization algorithm are defined. Rewriting a query into algebraic operators allows for the utilization of equivalence rules following a heuristic algorithm, reducing the execution cost of the query.

In order to reduce the number of computations required for query execution, appropriate equivalence expressions must be defined that allow for the removal of redundant computations in the query tree. Using these rules, a query algorithm may then be proposed to decrease the number of required computations for query execution, making the query more efficient.

Similar objectives and strategies have been employed in relational database contexts [4], and XML databases [5]. Our contribution to the current state of the literature is extending the propositions made in both of these contexts, in particular those related to query optimization heuristics, into a JSON context. A broad variety of JSON databases is utilized that allow us to make a fair comparison regarding both theoretical (quantitative dimension) as well as physical (running time) computational costs of a query, with and without the proposed optimization algorithm. We express our queries in JSONiq [6], which is a well-known and extensive JSON query language.

Based on our results, we conclude that our proposed algorithm, making use of our proposed data model, algebra, and equivalence rules, improves the execution of JSON queries both in terms of theoretical cost as well physical cost. Especially when dealing with databases that contain a large number of documents, a substantial difference can be observed in costs between executing a query with and without our optimization algorithm.

The remainder of this paper is structured as follows. Section 2 covers a literature review of related research, discussing current research progress regarding JSON data models, JSON query languages (QL), and JSON algebras. In Section 3 we describe the five databases that we use to test our optimization algorithm. In Section 4 we describe our methodology: we propose a JSON data model, define JSON algebraic operators, define equivalence expressions, and derive an optimization algorithm. In Section 5 we present our main findings. Last, in Section 6, we draw our conclusions and make suggestions for future research.

2 Related Work

This section begins with a discussion of JSON data models. Then, the current state of the literature regarding JSON query languages and JSON algebra is evaluated. Last, we discuss existing optimization approaches and their connections to our work.

2.1 JSON Data Model

Despite the prevalence of JSON in practical applications, there is no official standard for the modelling of JSON documents in the current state of the literature. In most JSON-related research, JSON documents are modelled by trees, i.e., JSON trees, where an important characteristic of the tree is that it is edge-labelled [7–9]. In [8], the structure of a simple JSON tree is described. This structure is especially useful when combined with the JSONPath query language, in which queries select the nodes of a tree where specific path conditions are met. In [9], the authors describe a JSON document through an object description that contains path-value pairs instead of the key-value pairs that form the main structure of JSON. A path is defined as the sequence of keys that leads to a specific value separated by dots, hence a path-value pair is similar to a key-value pair. Converting a JSON document into an object description results in a document like the one presented in Figure 1. In the object description, nesting is no longer present due to the path-value pairs. For example, the first name of a student represented in object description in Figure 1 is given by the path `name.first`. As keys assure that a JSON document is deterministic, it follows that a path is also well-defined. The presence of ordered data types, i.e., arrays, has

```
{
  "id"          : 123456,
  "name.first"  : "Jane",
  "name.last"   : "Smith"
  "courses.1"   : "EN",
  "courses.2"   : "SP",
  "courses.3"   : "MA",
  "courses.4"   : "GEO",
  "courses.5"   : "PHY",
  "courses.6"   : "SC",
  "courses.7"   : "CS",
  "courses.8"   : "PE"
}
```

Fig. 1: An example of a JSON document expressed in object description notation (paths followed by a colon and value)

not yet been physically tested in this approach. However, in [7], a suggestion is provided as to how to deal with arrays when modelling a JSON document as a JSON tree. The authors propose nodes for each element in an array where the edges are labelled with their respective index position in the array. Figure 2 shows a JSON tree that corresponds to the same JSON document as is depicted in Figure 1, displaying this idea of edges denoted by the position of an element in an array.

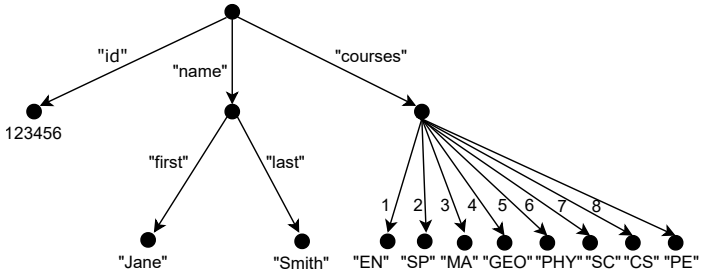


Fig. 2: An example of a JSON tree representation

2.2 JSON Query Languages

There are currently several query languages for JSON data:

1. JSONPath [10]
2. JAQL [11]
3. JSONiq [6]
4. JSONQuery [12]
5. JMESPath [13]
6. ObjectPath [14]
7. JSONata [15]
8. SQL++ [16]
9. J-CO-QL+ [17]

The query language of JSONPath [10] is based on XPath [18], a well-known XML QL. A notable power of this language is that it allows for the presence of arbitrary recursion in documents, which is not uncommon in JSON documents. However, it does not have official semantics and does not provide the ability to merge or join JSON documents [19]. Furthermore, it is only functional for selecting and extracting values from JSON, not providing any functions to create or update JSON documents. Should we want to extract the last name of the JSON document represented in Figures 1 and 2, the query would be expressed in JSONPath as: `$.name.last`, where `$` denotes the root object. Hence, JSONPath uses dot notation for its path navigation.

The next relatively well-known query language for JSON documents is JAQL, a QL originally devised by Google. It is mainly used for, but not limited to, the updating, querying and construction of large-scaled semi-structured data, i.e., JSON documents [11, 20]. Its main features are extensibility and parallelism, as it can not only run locally, but also in parallel utilizing Hadoop's Map-Reduce. Due to this scalability property, JAQL is mainly used in applications related to Big Data. JAQL does allow

for the presence of operators like joins and groupings [21], however, the join operation is limited to conditions with equality signs (e.g., joins with a condition containing the *larger than* logical operator are not allowed).

JSONiq is a JSON QL based on the fundamentals of SQL [4] and XQuery [22]. SQL is the most popular QL in relational databases and XQuery is a well-known, extensive, and productive QL for XML that is also platform-independent. The foundation behind JSONiq is the use of advancements from relational database systems and the utilization of semi-structured data. It allows for the construction, querying, and updating of nested, heterogeneous, and semi-structured data [6]. The construction of new JSON objects works by utilizing a syntax closely related to JSON itself. JSONiq has several useful properties [1]. Among others, it is a language that can process sets of JSON objects, consisting of possibly nested and heterogeneous data, at once, where expressions are fully composable. These expressions are the building blocks behind JSONiq. The expressions are ordered set-oriented, hence both input and output are sequences. The most powerful types of expressions in JSONiq are the `For, Let, Where, Order by, Return` (FLWOR) expressions. They are similar to SQL's `select-from-where` but are not bound to a particular order, except for the requirement that each either starts with a `for` or `let` clause and ends with a `return` clause [23]. There are seven available clauses in total, namely the `count`, `for`, `group by`, `let`, `order by`, `return`, and `where` clauses. It allows for, among others, selections, joins (of heterogeneous data), and projections. These expressions form the basis for query optimization. Further, like JAQL, JSONiq may also be run in parallel contexts, though we focus on its local capabilities and leave extensions to parallel computing as future research.

There are also lesser-known proposals. JSONQuery is an example of such, providing a superset of the functionality provided in JSONPath, extending it through operators such as mapping, sorting (not limited to array sorting as is the case in JSONPath), and the ability to perform inner-joins [12]. Two other JSON QLs that are derived from JSONPath are JMESPath, which is a JSON QL for processing (manipulating) and updating JSON data [13, 24] and ObjectPath, which again is a QL for processing JSON data [14]. Then, there is JSONata, a lightweight processing and updating language for JSON documents. It is extensible and can handle complex queries with minimal syntax [15]. In addition there is SQL++, which is a JSON QL that is to a large extent backwards compatible with SQL making it easy to use by people familiar with SQL [16]. Last, there is J-CO-QL [17], which is a JSON QL for geographical data represented in the J-CO platform [25]. In Table 1 we provide an overview of the functionalities supported by each of the aforementioned query languages. The functionalities we consider are the abilities to process JSON documents, create JSON documents, update a (value in a) JSON document, perform join-type operations, and sort a collection of JSON documents. *OK* signifies that the QL supports a given functionality, and an empty field indicates that the QL has no support for said functionality. An *L* indicates that there is limited support for the functionality. The most limited JSON QL is JSONPath, as join-type operations, creating JSON documents, and updating JSON documents are not allowed. To our knowledge, the JSON QL that is most expressive dedicated to semi-structured data (in the current state of literature) is JSONiq.

	Processing	Create	Update	Join	Sort
JSONPath	OK				L
JAQL	OK	OK	OK	L	OK
JSONiq	OK	OK	OK	OK	OK
JSONQuery	OK			L	OK
JMESPath	OK		OK		OK
ObjectPath	OK				OK
JSONata	OK		OK		OK
SQL++	OK	OK	OK	OK	OK
J-CO-QL+	OK			OK	OK

Table 1: Functionality overview for the nine discussed query languages

2.3 Optimization and JSON Algebraic Operators

Query optimization decreases a system's memory and storage usage as well as simultaneously reducing execution time while executing a query, all desirable effects in practical applications. Query optimization can be divided into two categories: logical optimization and physical optimization.

Logical optimization relates to the translation of a query into corresponding algebraic operators, defining conceptually how they operate and finding equivalent representations for said operators such that a query written solely in algebraic operators can be rewritten into a computationally cheaper form. Translating a query into its algebraic counterpart allows for its representation by a query tree [26]. In this tree, each leaf node represents the input data and each internal node refers to an algebraic operator. The root node of the tree denotes the query result. Utilizing logical optimization equivalence rules, a query tree can be rewritten without changing the root node, hence obtaining an identical output more efficiently. This concept is described in a relational context in [4].

An optimization algorithm utilizing heuristics and statistics can then be defined, where the most powerful heuristic is that of moving projections and selections as far down a query tree as possible. Performing these operations as early as possible reduces the magnitude of the intermediate results, in turn decreasing the number of required computations further up the tree. Heuristics, however, do not guarantee optimality. For example, if we are dealing with a multiple selection problem it is not always known which selection is the most restrictive in real-time. We can only make an approximation, as done in [27], through a selectivity heuristic.

The above concepts have been applied in an XML context in [5]. There, an algebra is proposed for XML query optimization specific to the XML data model. It is concluded that this strategy provides efficient transformations such that a significant reduction in the number of computations is possible. As XML describes semi-structured data, as does JSON, we explore the same strategy applied to a JSON context.

Physical optimization then relates to the physical implementation of each algebra operator. Physical optimization describes for each operator which algorithm should be

used for the implementation of the operator [4, 28]. There are several ways to implement each operator, and there is not generally a clear indication as to which results in the best performance. Therefore, in physical optimization one needs to compare these different implementations by their I/O costs dynamically.

In the current state of the literature, there are only a few (preliminary) specifications for algebraic operators applicable to JSON. PostgreSQL [29] offers several functions and operators for the processing and creation of JSON documents. In [30] several operators are also defined for the processing of JSON data. However, neither of these specifications offer any optimization rules nor an optimization algorithm. Queries of document based, non-relational databases have also been expressed in the form of relational algebra itself, such as in [31], where a benchmarking approach is proposed for use in gauging query performance in both relational and non-relational databases. This sort of query expression in relational algebra is, however, only possible for some JSON queries, with their non-relational nature generally preventing a direct relational representation. Additionally, other approaches to aid in the optimization of JSON queries in certain applications have been proposed, such as the JSON Tiles in [32]. There, the authors find that a more granular view of JSON document structure in relational contexts can allow for a more efficient evaluation of query expression costs, in turn enabling the construction of more efficient queries. This approach again, however, does not specify optimization rules, instead aiding in query optimization on existing relational engines with JSON support.

The main contributions of this work are as follows. To the best of our knowledge we are among the firsts to propose an algebra for JSON query optimization. After devising the algebra operators, we propose an algorithm for query optimization. While both algebra and optimization algorithm are inspired by the relational counterparts, they also contain elements specific to the semi-structured nature of JSON. Last, we have shown the usefulness of the algebra and optimization algorithm by applying these to query optimization for JSONiq, a popular and versatile query language for JSON, over several existing JSON databases.

3 Data

In this section the datasets, on which the proposed optimization algorithm is tested, are presented. Each subsection describes a database, for which we define the constituent datasets, the size of the datasets (in number of JSON documents), the keys within the datasets, as well as the schema of each document in the JSound schema language [33].

3.1 Dataset Sizes

The different datasets and the size of each dataset in bytes are provided in Table 2.

Case	Dataset	Bytes Size
Pokémon	Pokemons.json	81,997
Airports	AirportDelays.json	4,964,702
Bike sharing	station-information.json	26,277
	station-status.json	16,479
Rick and Morty	characters.json	358,734
	episodes.json	58,746
	locations.json	53,540
Nobel prize	Prizes.json	218,948
	Laureates.json	470,676
	Countries.json	4,615

Table 2: Dataset Sizes

3.2 Pokémon

Our first database relates to Pokémon, a Japanese media franchise starring mystical creatures. It contains the information of 151 Pokémon, hence there are 151 documents present in this database [34]. Each document contains the following keys:

- *id*: the id of the Pokémon;
- *name*: the name of the Pokémon;
- *height*: the height of the Pokémon in meters;
- *weight*: the weight of the Pokémon in kilograms;
- *weaknesses*: an array containing the weakness of the Pokémon (e.g., Fire).

JSound schema for documents in Pokemons.json

```
{
    "id": "integer",
    "name": "string",
    "height":"string",
    "weight": "string",
    "weaknesses": [ "string" ]
}
```

We refer to this collection in the queries using the name *pokemon*.

3.3 Monthly Airline Delays by Airport

The second database contains the monthly airline delays at airports in the United States [34]. The database includes data from the period beginning in 2003 up until and including 2016. This collection covers 4,408 documents, where each document contains the following keys:

- *Airport*: an object containing the *Name* and *Code* of the airport (e.g., an airport with *Name* "Los Angeles, CA: Los Angeles International" and with *Code* "LAX");

- *Time*: an object containing the keys: *Label*, *Month*, *Month Name*, and *Year* (e.g., October 2003 has as *label* "2003/10", *Month* 10, *Month Name* "October", and *Year* 2003);
- *Statistics*: an object containing another 3 objects:
  1. *# of Delays*: this object contains the keys *Carrier*, *Late Aircraft*, *National Aviation System*, *Security*, and *Weather*. Each key represents a cause for a delay. The object's values then denote how many delays were caused by the reason given in each key (e.g., 5 delays were caused due to a late aircraft).
  2. *Flights*: this object contains the keys *Cancelled*, *Delayed*, *Diverted*, *On Time*, and *Total*. Each key describes how many flights meet that category (e.g., 10 flights were diverted).
  3. *Minutes Delayed*: this object contains the keys *Carrier*, *Late Aircraft*, *National Aviation System*, *Security*, *Weather*, and *Total*. Each of their values represent how many minutes of delay were caused by that reason (e.g., 400 minutes of delay were caused by the weather).

JSound schema for documents in Airport Delays.json

```
{
    "Airport": {
        "Name": "string",
        "Code": "string"
    },
    "Time": {
        "Label": "string",
        "Month": "integer",
        "Month Name": "string",
        "Year": "integer"
    },
    "Statistics": {
        "# of Delays": {
            "Carrier": "integer",
            "Late Aircraft": "integer",
            "National Aviation System": "integer",
            "Security": "integer",
            "Weather": "integer"
        },
        "Flights": {
            "Cancelled": "integer",
            "Delayed": "integer",
            "Diverted": "integer",
            "On Time": "integer",
            "Total": "integer"
        },
        "Minutes Delayed": {
            "Carrier": "integer",
            "Late Aircraft": "integer",
```

```
            "National Aviation System": "integer",
            "Security": "integer",
            "Weather": "integer",
            "Total": "integer"
        }
    }
}
```

We refer to this collection in the queries using the name *airportMonth*.

3.4 Bike Share At Stations

Our third database contains information about bike sharing and their locations (at stations) [35]. The database consists of two datasets:

1. **Stations**, consisting of 95 JSON documents, each one related to a station. Each document contains the following keys:
   - *station_id*: the id of the station (e.g., "hub_299");
   - *name*: the name of the station;
   - *region_id*: the id of the region the station is located in (e.g., "region_80");
   - *address*: the address of the station.

JSound schema for documents in station-information.json

```
{
   "station_id": "string",
   "name": "string",
   "region_id": "string",
   "address": "string",
}
```

2. **Statuses**, consisting of 95 documents, hence each document contains the information of a single station. Each document contains the following keys:
   - *station_id*: the id of the station;
   - *num_bikes_available*: the number of bikes that are currently available for renting at the station;
   - *num_bikes_disabled*: the number of disabled bikes at the station;
   - *num_docks_available*: the number of bike storage racks currently available at the station.

JSound schema for documents in station-status.json

```
{
    "station_id":  "string",
    "num_bikes_available":  "integer",
    "num_bikes_disabled":  "integer",
    "num_docks_available": "integer",
}
```

In the queries we refer to these 2 collections using the names *station* and *status* for the **Stations** and **Statuses** datasets, respectively.

3.5 Rick and Morty

Our fourth database is related to the cartoon TV-show "Rick and Morty" [34]. This database contains the following three datasets:

1. **Characters**, consisting of 671 JSON documents, each one related to a character of the cartoon. For each character we may know the following information:
   - *id*: the id of the character;
   - *name*: the name of the character;
   - *url*: a string containing a url refering to a website containing more information about the character;
   - *status*: denoting the status (e.g., dead or alive) of the character;
   - *species*: denoting what type the character is;
   - *gender*: the gender of the character;
   - *origin*: an object containing the *name* of the character's birthplace and a *url* referring to a website containing more information about the origin location;
   - *location*: an object containing the *name* of the character's current location and a *url* referring to a website containing more information about the location.

JSound schema for documents in characters.json

```
{
    "id": "integer",
    "name": "string",
    "url": "anyURI",
    "status": "string",
    "species": "string",
    "gender": "string",
    "origin": {
        "name": "string",
        "url": "anyURI"
    },
    "location": {
        "name": "string",
        "url": "anyURI"
    }
}
```

2. **Episodes**, consisting of 41 JSON documents, each one related to an episode. For each episode we may know the following information:
   - *id*: the id of the episode;
   - *name*: the name of the episode;
   - *episode*: the episode code (e.g., "S01E01");
   - *characters*: an array containing the *url*'s for each character that appears in the episode;

- *url*: a string containing a url referring to a website that contains more information about the episode.

JSound schema for documents in episodes.json

```
{
  "id": "integer",
  "name":  "string",
  "episode":  "string",
  "characters": [ "anyURI" ],
  "url": "anyURI"
}
```

3. **Locations**, consisting of 108 JSON documents, each one related to a location. For each location we may know the following information:

   - *id*: the id of the location;
   - *name*: the name of the location;
   - *type*: the type of location (e.g., "Planet");
   - *url*: a string containing a url referring to a website containing more information about the location.

JSound schema for documents in locations.json

```
{
    "id": "integer",
    "name": "string",
    "type": "string",
    "url": "anyURI"
}
```

In the queries we refer to these 3 collections using the names *character*, *episode*, and *location* for the **Characters**, **Episodes**, and **Locations** datasets, respectively.

3.6 Nobel Prize

Our fifth database describes all awarded Nobel prizes and their laureates within the period from 1901 until 2020 [34]. This database consists of three datasets:

1. **Prizes**, consisting of 652 JSON documents, each one related to a prize. Each document contains the following keys:

   - *year*: the year the prize was awarded;
   - *category*: the research field the prize is awarded in;
   - *laureates*: an array of objects covering the winners of the prize, where each object contains the *id* and *motivation* of the winner.

JSound schema for documents in Prizes.json

```
{
```

```
    "year": "string",
    "category":  "string",
    "laureates": [
        {
          "id": "string",
          "motivation": "string"
        }
    ]
}
```

2. **Laureates**, consisting of 955 JSON documents, each one related to a laureate. Each document contains the following keys:

   - *id*: the id of the laureate;
   - *firstname*: the laureate's first name;
   - *surname*: the laureate's surname;
   - *bornCountryCode*: the country code of the country the laureate was born in;
   - *diedCountryCode*: the country code of the country the laureate died in;
   - *gender*: the gender of the laureate.

JSound schema for documents in Laureates.json

```
{
    "id": "string",
    "firstname": "string",
    "surname": "string",
    "bornCountryCode": "string",
    "diedCountryCode": "string",
    "gender": "string"
}
```

3. **Countries**, consisting of 135 JSON documents, each one related to a country. Each document contains the following keys:

   - *code*: the country code;
   - *name*: the name of the country.

JSound schema for documents in Countries.json

```
{
    "code": "string",
    "name": "string"
}
```

In the queries we refer to these 3 collections using the names *prize*, *laureate*, and *country* for the **Prizes**, **Laureates**, and **Countries** datasets, respectively.

4 Methodology

This section describes our methodology, beginning with a brief description of JSON and a formal description of our JSON data model, as well as the underpinnings and basic definitions of our JSON Algebra, JAL. We then expand on the key equivalences that enable optimization and last describe our actual optimization approach.

4.1 JSON Data Model

```
{
  "id"      : 123456,
  "name" : {
              "first": "Jane",
              "last": "Smith"
            },
  "courses": ["EN", "SP", "MA", "GEO", "PHY", "SC", "CS", "PE"]
}
```

Fig. 3: An example of a JSON document

We begin with an overview of JSON and its general structure. JSON is a light-weight, schema-less, human-readable, and machine-readable data-interchange format built as a subset of the JavaScript Programming Language (using conventions seen in JavaScript). JSON can be characterised by two data structures: objects and arrays. Objects are unordered sets of key-value pairs, where each key should be of type string and the corresponding value can be any of the following data types: string, number, array, Boolean, null, or again an object. Each object is denoted by curly brackets with data-fields as key-value pair(s) inside the brackets: $\{key: value\}$. Note that each JSON document is enclosed by these brackets as well, as a JSON document is itself an object. The second data structure encountered in JSON documents are arrays. These ordered sequences of values can again contain a mix of all the aforementioned data types, e.g., $[value_1, value_2, value_3]$, where, for instance, $value_1$ is an integer, $value_2$ is a string, and $value_3$ is an object. A simple example of a JSON document can be seen in Figure 3. The document contains the id, denoted by a number, the name of the student, split into an object containing the first and last name, and the courses the student follows denoted as (string) abbreviations contained in an array. These standard data structures simplify the process of machine reading of documents, especially useful from the perspective of data-interchange.

We now describe the formal specification of our JSON data model. From Section 2.1, it can be concluded that the literature does not know a standard for a JSON data model, although most formalizations share similarities between them. The JSON data model presented in this paper is inspired by the JSON data models and notation described in [5, 7–9, 36]. Each JSON document d is in itself a JSON object and each

JSON object can be represented by a tree-shaped structure. A tree is a connected, directed, and acyclic graph. A JSON tree $\mathcal{T}$ is an edge-labelled tree, meaning that the edges are labelled with the keys from a JSON document and the nodes represent the corresponding values. Following JSON's compositional structure, each node, n, has exactly one parent node except for the *root* node that denotes the entire document d. The formal definition of the JSON data model is as follows:

$$\mathcal{T} = (root, \mathcal{E}, \mathcal{N}), \text{ where } root = d$$

$$\mathcal{E} = E_{str} \cup E_{int}$$

$$\mathcal{N} = N_{internal} \cup N_{leaf}$$

A JSON tree $\mathcal{T}$ consists of three components: a *root* node, a set of edges $\mathcal{E}$, and a set of nodes $\mathcal{N}$. The set $\mathcal{E}$ represents two types of edges, namely the string edges (E_{str}) and the integer edges (E_{int}). The former denotes the edges that are labelled with a key k. As keys in JSON documents are always of type string, each edge e from this set is of type string, that is $\forall e \in E_{str}, type(e) = string$. The set E_{int} then denotes the edges that are labelled with an integer value. The inclusion of such edges is required for the representation of an array value in a tree. As order is not present in a tree, arrays can not be directly represented in a tree. Therefore, elements of an array are represented by nodes, where each node can be accessed by an edge labelled with its corresponding position i in the array, e.g., the edge leading to the first element $index = 1$ of an array will be labelled with the number 1 (e.g., the manner in which the courses are displayed in Figure 2). Naturally, it follows that each edge e from this set is of type integer, that is $\forall e \in E_{int}, type(e) = \mathbb{N}$. Hence, the values of the natural numbers themselves are not important, only the order they represent is. Each edge e has five properties, as can be seen in Table 3. Each property can be accessed by an accessor function, denoted by their property name (e.g., the function $label(e)$ returns the label belonging to edge e).

Property	*Definition*
label	the name of the edge, either a key or a number
parent	the source node of the edge
child	the target node of the edge
type	the type of edge
identifier	a unique number that refers only to a particular edge

Table 3: Properties for an edge in a JSON tree

The set $\mathcal{N}$ represents two types of nodes, namely the leaf nodes (N_{leaf}) and the internal nodes ($N_{internal}$). A leaf node represents an atomic value v. Atomic values are defined as:

- **string**: $v =$ "s", if "s" is a sequence of Unicode characters;
- **numbers**: $v = m$, if $m \in \mathbb{D}$, the set of decimal numbers;

- **boolean**: $v = b$, if $b \in \{true, false\}$;
- **empty**: $v = \varnothing$, if $\varnothing$ represents "null".

The internal nodes represent complex values. Complex values are the two other data types that JSON documents allow: arrays and objects. Formally, complex values are defined as:

- **arrays**: $v = [v_1, v_2, \ldots, v_{|v|}]$, hence an array consists of $|v|$ elements where each element of v can either be an atomic value or again a complex value;
- **object**: $v = \{k_1 : v_1, k_2 : v_2, \ldots, k_{|v|} : v_{|v|}\}$, hence an object consists of $|v|$ key-value pairs where each key k is of type string and each value v is either an atomic value or again a complex value.

Thus, complex values allow for the presence of an arbitrary level of nesting, hence the name complex values. Each node has four properties, depending on its type, as can be seen in Table 4. The *outgoingEdges* property is required when the node *type* is complex. Atomic nodes can not have *outgoingEdges*. The *value* of a node depends on its *type*: if the node is complex, the *value* of the node is equal to its *identifier* and if the node is atomic, the *value* of the node is equal to its actual atomic value (i.e., a string, boolean, number, or empty). An accessor function can access each property. Again the accessor functions are denoted by their property name.

Property	*Definition*
value	the value represented by the node
outgoingEdges	the edges that go out of the node
type	the type of node
identifier	a unique number that refers only to a particular node

Table 4: Properties for a node in a JSON tree

If there are several JSON trees, $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n$, then we speak of a collection of trees. To simplify notation, we sometimes identify a tree by its root node (the nodes and edges accessible from this root node form the set of nodes and set of edges associated with the tree), allowing us to speak of collections of nodes. Formally, a collection $\mathcal{C}$ is defined as follows: $\mathcal{C} = [n_1, \ldots, n_{|\mathcal{C}|}]$, where $|\mathcal{C}|$ denotes the number of nodes in the collection $\mathcal{C}$. The nodes $n_i, i = 1, ..., |C|$, could be a leaf node or an internal node. Note that a collection is in essence an array type of structure, hence order of the nodes is technically present. However, as we are dealing with JSON data, this order can be ignored in contexts where order is unnecessary.

4.2 JAL: a JSON Algebra

As we are dealing with schema-less, heterogeneous data, there are several requirements which must be in place to be able to process such data: basic operators should be robust to variations in data types and recursion should be present to deal with arbitrary nesting in the JSON objects. Here we are inspired by [5], whose XML algebra (XAL) addressed these same concerns in the context of XML query optimization.

We distinguish 3 types of operators: extraction, meta, and construction operators. Extraction operators cover a set of operators that can retrieve specific information from a node. Meta-operators deal with controlling the evaluation of the operators. Construction operators are responsible for the creation of new nodes, edges, or even documents. Important to note is that operators can either be unary or binary. As binary operators allow for the comparison of two nodes, node equality needs to be defined. Two leaf nodes are considered equal if and only if their node *value* is equal, with leaf nodes not possessing a unique identifier. Two internal nodes, on the other hand, are considered equal if they have the same identifier. The binary operators are defined similarly as to those seen in relational algebra. For optimization purposes, all binary operators are required to have multiset variants such that commutativity ($a \times b = b \times a$) can be used, where order is not important. These binary operators are not set operators, but multiset operators, as their operands are unordered collections where duplicates are allowed. We may thus choose to ignore the ordering present in the JSON data model of Section 4.1 for the sake of improved query optimization.

We define a general form for both unary as well as binary operators. They are given as follows:

Unary operator general form: $o[f](x : expression)$

Binary operator general form: $(x : expression)\ o[f]\ (y : expression)$,

The $(x : expression)$, and similar $(y : expression)$, components retrieve each node (x or y) following from a collection characterized by an *expression*, hence they form the input for the operator. An *expression* can be an entire collection or a query involving an arbitrary composition of operators. This is possible due to the closedness property of the operators, i.e., the input of an operator is a collection (or set of collections) and the output is as well. The $o[f]$ component computes (based on the semantics of o) for each node the partial result of applying f and concatenates the obtained results to each other, hence it follows that these general forms are another way of denoting a collection. The function f operates on nodes. Note that f is optional in case o operators act directly on x, y. As mentioned in [5], this notation is similar to list comprehensions seen in functional programming and monads seen in mathematics. A list comprehension is given in the general form [37]:

$$[< expression > \mid < qualifier >; \ldots; < qualifier >]$$

where the output is an array. The *expression* defines the characterizing properties for the output, or put differently, it transforms the data into the desired output, e.g., if that is a Cartesian product of two collections, an expression will take on the form (x, y). The x and y follow from the $qualifiers$, more specifically from the *generator* type of $qualifier$. The other type of $qualifier$ is a $filter$ that puts additional conditions on the data from the *generator* (e.g., only keep the even values from the *generator*). A monad is a triplet of functions over a category (which is comparable with a collection) defined generally as [38]:

$$(M, \eta, \mu)$$

where M is similar to XAL's f function (which comes down to being a map operator), η defines how the partial result is returned, and μ can be seen as the aggregator of the partial results. These similarities allow us to use equivalent expressions proposed for both concepts.

Our operators are defined as follows, where each example is based on the running example in this paper, hence a collection $\mathcal{C}$ with one JSON tree like the one given in Figure 2:

A: Extraction operators

1. **Projection**
   The projection operator is used to reduce a collection to another collection of specific data. The operator takes as input a collection of nodes , returning as output a collection of nodes with edges that are annotated with the given label(s) and refer to the original targets.

   The general form of the projection operator is given by:

   $$\pi_{[label]}(x : expression)$$

   where $[label]$ is an array composed of one or more labels.
   Example:
   If we want to extract all nodes linked solely to edges with label id from the collection $\mathcal{C}$, we can employ: $\pi_{[id]}(x : \mathcal{C})$. This operator will return a new collection consisting of nodes which have edges labelled id and their original targets.
2. **Selection**
   The selection operator reduces a collection to a subcollection of itself. However, instead of selecting data based on edge labels like the projection operator, a selection operator uses a selection condition c and returns only the data that fulfills said condition. The general form of the selection operator is defined as:

   $$\sigma_c(x : expression)$$

   A condition c can be formulated as a statement starting with a *label*, followed by a comparison operator p and a *value*, where $p \in \{=, \neq, >, \geq, <, \leq\}$. To formulate a more complex condition, one or more logic operators l can be used, $l \in \{\wedge, \vee, !\}$. We limit ourselves to these operators and leave the inclusion of a more complete set of operators (e.g., arithmetic operators) to future research.
   Example:
   If we want to extract all nodes that have an attribute id with the value *123456*, we may utilize: $\sigma_{id=123456}(x : \mathcal{C})$.
3. **Unorder**
   If a collection is sorted and it is desired to transform the collection into an unordered one, the unorder operator can be utilized. After applying the unorder operator, a collection is randomly ordered. The general form of the unorder operator is defined as:

   $$\chi(x : expression)$$

Example:
If we are dealing with an input collection $\mathcal{C} = [n_1, n_2, n_3]$, a potential output collection resulting from use of the unorder operator may appear as: $\chi(x : \mathcal{C}) = [n_2, n_1, n_3]$. While the output collection still has an ordering present, any meaning behind said ordering is lost and the collection may now be treated as unordered.

4. **Distinct**
   As described in Section 4.1, a collection has an array structure allowing for the presence of duplicate nodes within a single collection. The distinct operator removes such duplicates while maintaining the input order, preserving the first instance of any duplicate nodes and discarding all subsequent instances. Distinctions are made between nodes in the input collection on the basis of a unique identifying value found by following the top-level edge indicated by a single pre-specified *label*. The general form of the distinct operator is defined as:

$$\delta_{label}(x : expression)$$

   Example:
   If we are dealing with an input collection, $\mathcal{C} = [n_1, n_2, n_1]$, the output collection following from the distinct operator will be: $\delta_{id}(x : \mathcal{C}) = [n_1, n_2]$. The unique identifying attribute in this case is *id*, which in our running example refers to a student number.
5. **Sort**
   The sort operator orders a collection based on some *value_expression*. The general form of the sort operator is:

$$\Sigma_{value\_expression(x)}(x : expression)$$

   where the $value\_expression(x)$ can be defined as any function operating over some node x and returning a value. If the sole input of this function is simply a set of leaf nodes, the collection will be sorted according to their values without needing to specify a $value\_expression(x)$. If the same value is encountered more than once, the nodes are ordered according to their relative order in the input collection (the system order is used for unordered collections).
   Example:
   The output for the sort operator $\Sigma_{\pi_{[last]}(x)}(x : C)$ orders all nodes alphabetically based on the child node value targeted by the edge with label *last*, hence it orders all nodes based on the last name of a student (we assume the input collection has nodes with the label *last*).
6. **Join**
   A join operator combines data that meet a specific condition c from two collections into a single collection. The join operator is defined as follows:

$$(x : expression) \bowtie_c (y : expression)$$

   For each node of the first collection and each node of the second collection, selected by way of an external loop through the first collection and internal loop through the second collection, a new joined node is created and added to the output collection.

This is done by first creating a new root node, to which all the edges of the node from the first collection are added, before adding the edges of the node from the second collection. Once all joined nodes have been created from all pairs of nodes from the input collections, the condition c is then applied on the resulting collection in the same manner as in the selection operator. Should the input collections be unordered, the multiset variant of this operator is applied with a resulting output collection that is similarly unordered. If, however, the input collections are ordered, the output will be similarly ordered, following the order of the first input collection.
Example:
Suppose we have the collection of students $\mathcal{C}_1$ (the running example) and a new collection of average grades $\mathcal{C}_2$. The latter collection contains a node for each student containing their average grades and their id. A join of these collections, conditioned on the id attribute, will look like: $(x : \mathcal{C}_1) \bowtie_{\pi_{[id]}(x:\mathcal{C}_1)=\pi_{[id]}(y:\mathcal{C}_2)} (y : \mathcal{C}_2)$.

7. **Cartesian Product**
   A Cartesian product is the cross-product of all possible node pairs based on two input collections. It is a special case of the join operator, where the join condition is always true. The Cartesian product operator is defined as follows:

$$(x : expression) \times (y : expression)$$

8. **Union**
   A union operator merges data from two collections into a single collection. The data does not have to meet a specific condition like we have seen for the join operator. The inputs for the union operator are two collections, with as output a single collection containing all nodes (and edges) from both collections. The general form of a union operator is defined as follows:

$$(x : expression) \cup (y : expression)$$

   As with the join (and by extension Cartesian product) operator, should the union operator be applied to unordered input collections, a multiset variant is used which results in an unordered output. Should the inputs be ordered, the output collection is ordered as the first input collection followed by the second input collection, maintaining the order within each.
   Example:
   Suppose we have two collections $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{C}_1 = [n_1, n_2, n_3]$ and $\mathcal{C}_2 = [n_3, n_4, n_5]$. The union operator $(x : \mathcal{C}_1) \cup (y : \mathcal{C}_2)$ will then result in $[n_1, n_2, n_3, n_3, n_4, n_5]$.
9. **Intersection**
   An intersection operator finds the common nodes between two collections. The input for this operator are two collections, returning a single collection as output. The general form of an intersection operator is defined as:

$$(x : expression) \cap (y : expression)$$

   As with the other binary operators, a multiset variant of the intersection operator is used when the input collections are unordered, resulting in an unordered output. In

the case of ordered input collections, the output collection maintains the ordering of the first input collection.
Example:
Suppose we have two collections $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{C}_1 = [n_1, n_2, n_3]$ and $\mathcal{C}_2 = [n_3, n_4, n_5]$. The intersection operator $(x : \mathcal{C}_1) \cap (y : \mathcal{C}_2)$ will then result in $[n_3]$.

10. **Difference**
    A difference operator returns all nodes that are present in the first collection but are not present in the second collection. It takes as input two collections and outputs a subcollection of the first collection. The general form of a difference operator is:

$$(x : expression) - (y : expression).$$

    The difference operator behaves the same way as the intersection operator with regard to ordering. Should the input collections be unordered, the output is unordered; if the input collections are ordered, the output collection follows the order of the first input collection.
    Example:
    Suppose we have two collections $\mathcal{C}_1$ and $\mathcal{C}_2$, $\mathcal{C}_1 = [n_1, n_2, n_3]$ and $\mathcal{C}_2 = [n_3, n_4, n_5]$. The difference operator $(x : \mathcal{C}_1) - (y : \mathcal{C}_2)$ will then result in $[n_1, n_2]$.

B: Meta-operators

1. **Map**
   A map operator applies a specific function f to each tree in a collection. It takes as input a collection and the function f, where f can be any unary operator, and returns as output a modified input collection by applying f to each element of the input collection. The general form of the map operator is given by:

$$map_f(x : expression)$$

   Example:
   $map_{identifier}(x : C)$ where C represents the collection of all nodes in the input, determines the identifiers of all the nodes in the input, i.e., the values referred by the $identifier$ property.

2. **Kleene Star**
   A Kleene Star operator repeats a specific function an arbitrary number of times on a collection. It takes as input a collection and some function f. The general form of the Kleene Star operator is given by:

$$*_f(x : expression) = x + f(x) + f(f(x)) + f(f(f(x))) + ...$$

   The recursion can be of arbitrary depth, as termination is reached when an iteration's output is equal to its input. If it is desired to assure that the operator terminates after a certain amount of iterations, a pre-defined number q, $q \in \mathbb{N}$, can be passed on to the operator (i.e., $*_{f,q}(x : expression)$).
   Example:
   Suppose that the collection described in our running example is extended with some

arbitrary number of trees with a similar structure as the first tree, hence each tree describes a student profile. Additionally, assume that each tree can contain an edge labelled *parent* that points to a node(s) that describes that student's parent(s) that also attend or have attended the same school, each of which may themselves have a parent. The names of the parents can then be retrieved using the Kleene Star operator as follows: $\pi_{[name]}(*_{\pi_{[parent]}(x)}(x : \mathcal{C})$.

C: Construction operators

1. **Node**
   The node operator creates a new node, taking as input the *value* and *type* of the node. Note that it is possible to create a node of complex type. If the node to be created is complex, we use as *value* the id returned by the new id generator $nig()$, which returns a new node id in the current context. The general form of the node operator is defined as:
   $$node_{[value,type]}()$$
   Example:
   Suppose we want to construct a new atomic node. Then $node_{[`EUR',atomic]}()$ creates an atomic node with the value 'EUR'.
2. **Edge**
   The edge operator creates a new edge. The operator takes as input a a *label*, *parent* node, *type*, and *child* node. The general form of the edge operator is:
   $$edge_{[label,parent,type]}(child)$$
   Example:
   Suppose we have constructed the node from the previous example, and we would like to add an edge between this node and some parent node n. As the node contains the name of a school, the label on the edge should be equal to the string '*school*'. Hence we construct an edge with label '*school*', with parent node n, child node n_c, and that is of type *string*. Thus this is defined as $edge_{[`school',n,string]}(n_c)$.
3. **Copy**
   Just as in XAL, we permit copying an edge, node, or even an entire tree. The copy operator depends on the node and edge creation operators presented above. To further clarify this dependency, we present the following examples:
   Copying a node:
   Suppose we want to copy the node presented in the example for **node creation** and suppose this node is called n. This would then look like $node_{[value(n),type(n)]}()$, where $value()$ and $type()$ are accessor functions. Hence it is simply the node operator as seen in the aforementioned example.
   Copying an edge:
   Suppose we want to copy the edge presented in the example for **edge creation** and suppose this edge is called e, with child node n. This will be possible through the operator: $edge_{[label(e),parent(e),type(e)]}(n_c)$, where $label(), parent()$, and $type()$ are accessor functions.

4.3 JSON Algebra Equivalent Expressions

In this section, several equivalent expressions for the defined algebraic operators are presented. The equivalence expressions aim to enable a more efficient execution of a given expression. The rules are based upon the propositions made in [4]. There, equivalence rules are defined in a relational context. However, in [5] it is shown that these rules can be extended into the non-relational domain.

Rule 1: *Cascade of* σ

$$\sigma_{c_1 \wedge c_2 \wedge \cdots \wedge c_n}(\mathcal{C}) = \sigma_{c_1}(\sigma_{c_2}(\ldots \sigma_{c_n}(\mathcal{C}) \ldots))$$

Rule 2: *Commutativity of* σ

$$\sigma_{c_1}(\sigma_{c_2}(\mathcal{C})) = \sigma_{c_2}(\sigma_{c_1}(\mathcal{C}))$$

Rule 3: *Cascade of* π
If $label_{i-1}$ is a subset of $label_i$ for $i = 2, ..., m$, then

$$\pi_{[label_1]}(\pi_{[label_2]}(\ldots \pi_{[label_m]}(\mathcal{C}))) = \pi_{[label_1]}(\mathcal{C})$$

Rule 4: *Commuting of* σ *with* π
If the condition c solely requires nodes that are targeted by the labels in $label$, then

$$\pi_{[label]}(\sigma_c(\mathcal{C})) = \sigma_c(\pi_{[label]}(\mathcal{C}))$$

Rule 5: *Commuting of* σ *with* $\times$
If all the labels in the selection condition c are strictly of $\mathcal{C}_1$, then

$$\sigma_c(\mathcal{C}_1 \times \mathcal{C}_2) = \sigma_c(\mathcal{C}_1) \times \mathcal{C}_2$$

Rule 6: *Commuting of* $\cup$, $\cap$, $\bowtie$, *and* $\times$
If the ordering of collections is ignored, and therefore, multiset usage of binary operators is allowed, then

$$\mathcal{C}_1 \cup \mathcal{C}_2 = \mathcal{C}_2 \cup \mathcal{C}_1$$

$$\mathcal{C}_1 \cap \mathcal{C}_2 = \mathcal{C}_2 \cap \mathcal{C}_1$$

$$\mathcal{C}_1 \bowtie_c \mathcal{C}_2 = \mathcal{C}_2 \bowtie_c \mathcal{C}_1$$

$$\mathcal{C}_1 \times \mathcal{C}_2 = \mathcal{C}_2 \times \mathcal{C}_1$$

Rule 7: *Commuting of* σ *with* $\cup, \cap$, *or* $-$
If all labels in the selection condition c are strictly of $\mathcal{C}_1$, then

$$\sigma_c(\mathcal{C}_1 \cup \mathcal{C}_2) = \sigma_c(\mathcal{C}_1) \cup \mathcal{C}_2$$

$$\sigma_c(\mathcal{C}_1 \cap \mathcal{C}_2) = \sigma_c(\mathcal{C}_1) \cap \mathcal{C}_2$$

$$\sigma_c(\mathcal{C}_1 - \mathcal{C}_2) = \sigma_c(\mathcal{C}_1) - \mathcal{C}_2$$

Rule 8: *Commuting of* π *with* $\cup$

$$\pi_{[label]}(\mathcal{C}_1 \cup \mathcal{C}_2) = \pi_{[label]}(\mathcal{C}_1) \cup \pi_{[label]}(\mathcal{C}_2)$$

Rule 9: *Selection on a Cartesian product can be written as a conditional join*

$$\sigma_c(\mathcal{C}_1 \times \mathcal{C}_2) = \mathcal{C}_1 \bowtie_c \mathcal{C}_2$$

Rule 10: *Associativity of* $\bowtie, \cup, \cap$, *and* $\times$

$$(\mathcal{C}_1 \bowtie_c \mathcal{C}_2) \bowtie_c \mathcal{C}_3 = \mathcal{C}_1 \bowtie_c (\mathcal{C}_2 \bowtie_c \mathcal{C}_3)$$

$$(\mathcal{C}_1 \cup \mathcal{C}_2) \cup \mathcal{C}_3 = \mathcal{C}_1 \cup (\mathcal{C}_2 \cup \mathcal{C}_3)$$

$$(\mathcal{C}_1 \cap \mathcal{C}_2) \cap \mathcal{C}_3 = \mathcal{C}_1 \cap (\mathcal{C}_2 \cap \mathcal{C}_3)$$

$$(\mathcal{C}_1 \times \mathcal{C}_2) \times \mathcal{C}_3 = \mathcal{C}_1 \times (\mathcal{C}_2 \times \mathcal{C}_3)$$

Rule 11: *Empty collection*
Any operator over an empty collection returns an empty collection.

Rule 12: *Decomposition of* π

$$\pi_{[label_1, label_2, \ldots label_n]}(\mathcal{C}) = \pi_{[label_1]}\mathcal{C} \cup \pi_{[label_2]}\mathcal{C} \cup \cdots \cup \pi_{[label_n]}\mathcal{C}$$

Rule 13: *Commuting of* π *with* $\times$
If $label_1$ strictly contains labels belonging to $\mathcal{C}_1$ and $label_2$ strictly contains labels belonging to $\mathcal{C}_2$, then

$$\pi_{[label_1, label_2]}(\mathcal{C}_1 \times \mathcal{C}_2) = \pi_{[label_1]}(\mathcal{C}_1) \times \pi_{[label_2]}(\mathcal{C}_2)$$

Rule 14: *Commuting of σ with $\bowtie$*
If all labels in the selection condition are strictly of $\mathcal{T}_1$, then

$$\sigma_{c_1}(\mathcal{C}_1 \bowtie_{c_2} \mathcal{C}_2) = \sigma_{c_1}(\mathcal{C}_1) \bowtie_{c_2} \mathcal{C}_2$$

Rule 15: *Decomposition of π over $\bowtie$*
If $label_1$ strictly contains labels belonging to $\mathcal{T}_1$ and $label_2$ strictly contains labels belonging to $\mathcal{T}_2$, then

$$\pi_{[label_1, label_2]}(\mathcal{C}_1 \bowtie_c \mathcal{C}_2) = \pi_{[label1]}(\mathcal{C}_1) \bowtie_c \pi_{[label2]}(\mathcal{C}_2)$$

4.4 JSON Query Optimization Algorithm

Based on the above equivalence expressions, a heuristic optimization algorithm can be defined to rewrite parts of a query and make the query more efficient from the point of view of execution. A query can internally be represented by a query tree, where leaf nodes represent the input data and internal nodes represent the algebraic operators, hence a query tree is executed bottom-up. This structure allows a heuristic algorithm to interchange nodes and thus optimize the way a query is executed while guaranteeing that the query output will be identical to the output without optimization. In [4] principles for algebraic manipulation and an optimization algorithm for relational expressions are proposed. The principles should be interpreted heuristically as optimality can not be guaranteed. These principles are also recurring in the optimization algorithm proposed in [5]. Both these algorithms, as well as [39, 40], form the inspiration behind the following optimization algorithm:

Algorithm 1 JAL Query Optimization

INPUT: A JAL Query
OUTPUT: An Optimized JAL Query
1: *Remove empty operations using Rule 11.*
2: *Decompose joins using Rule 9.*
3: *Separate any selections with conjunctive conditions and move them as far down into the query tree as possible using Rules 1, 2, 4, 5, 7, and 14.*
4: *Perform the most restrictive selections (smallest output) as early as possible using Rules 6 and 10.*
5: *Decompose projections and move projections as far down the query tree as possible using Rules 3, 8, 12, 13, and 15.*
6: *Combine cascades of selections and projections into a single selection, single projection, or a selection followed by a projection.*

The primary heuristic of this algorithm is reducing the size of the intermediate output as early as possible. Therefore, the order of evaluation should be flexible, which requires the operators to be commutative. Projections and selections should be moved

down the query tree as much as possible, thus reducing the number of iterations further up the tree. Down the tree, the selection and projection operations that are most restrictive are to be performed first, thus reducing the amount of memory and storage required to perform a query while still returning the correct query output. The most restrictive projection is the projection that keeps the fewest edges, hence resulting in the smallest JSON document. As there is no way of knowing what the most restrictive selections will be in real-time, we propose the following heuristic rules for the determination of the restrictivity of a condition, from most restrictive to least restrictive: Furthermore, if there are several conditions that fall into the same category,

Algorithm 2 JAL Conditions Restrictivity

1: *If a condition has the comparison operator* $=$ *and is only bound to comparisons over one collection, we consider this condition as most restrictive.*
2: *If a condition is only bound to comparisons over one collection, and the comparison is not* $=$*, but one of the other operators* $\{\neq, >, \geq, <, \leq\}$.
3: *If a condition makes comparisons over more than one collection, and the comparison is* $=$.
4: *If a condition makes comparisons over more than one collection, and the comparison operator is not* $=$*, but one of the other operators* $\{\neq, >, \geq, <, \leq\}$*, we consider this condition to be the least restrictive condition.*

shorter conditions take precedence over longer conditions. This means that if, for example, we have a condition containing a single comparison and a condition composed of several *or* statements, all containing the equal comparison operator, the former is considered more restrictive than the latter. Also of note is that we make use of general comparison semantics in all instances, evaluating a comparison condition between two sequences to be true if at least one element of the first sequence satisfies the condition when compared with at least one element of the second sequence.

4.5 Use Cases

To validate the proposed heuristic optimization algorithm, we execute several different queries with and without the proposed heuristic. Comparing these queries in terms of theoretical and physical cost then displays the benefits of optimization compared to an unoptimized baseline. The theoretical cost denotes the quantitative dimension created by the query (i.e., how many elements are created through the presence of a join-type operation), as seen in [5]. The physical cost is expressed as the execution time needed to execute the query. Each query is formulated in some high-level language. In this paper, we solely use the JSONiq query language for the formulations of queries. The reasoning behind this decision is three-fold. Firstly, JSONiq supports FLWOR expressions that bear significant similarity with operators used in the algebraic context. Secondly, JSONiq's primary focus is querying, which allows for high optimizability compared to the other query languages mentioned in Section 2.2. Last, it is the JSON variant of XQuery, which is the QL used in [5], which inspired our work.

Lexer Rules	
Rule Name	**Rule** (written in RegEx)
VARIABLE ASSIGNMENT	*let \$.+(:=)*
PROJECTION	*for \$.+? in .+?(?=for|where|return|for|let|order by)*
CONDITION	*where \$.*?(?=for|return|for|let|order by|where)*
RETURN	*return .+?(?=return)*
SORT	*order by .*?(?=return|for|let|order by)*

Table 5: The grammar used in our lexer for the lexical analysis of JSONiq queries

Starting with a JSONiq query, the query is then translated to its JSON algebraic counterpart, such that the heuristic optimization algorithm can be applied to this intermediate form of the query. This is done by a compiler consisting of a lexing and a parsing component. For the lexer function we adjusted a public domain lexer, while we built the parser ourselves. For the lexical analysis of the JSONiq queries, we specified a grammar consisting of 6 rules. These rules are expressed in RegEx notation and capture the semantics of a JSONiq query that are relevant for the translation to JAL. These rules are formulated in Table 5, with a further elaboration in Appendix A.

The parser returns a query tree that can be executed and optimized. The processing of an optimized query can be seen in Figure 4. Each query is run with and without

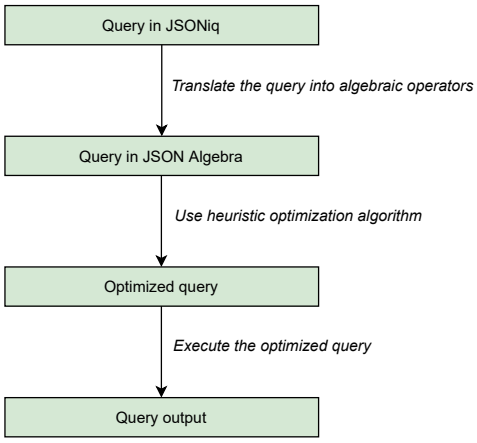


Fig. 4: Processing of a query with optimization

optimization in order to compare its theoretical (based on the number of joins) and physical (based on execution time) costs. We make comparisons based on several databases that vary in total database size and the number of intermediately linked keys. We expect that complex queries on large-scaled data will have a significantly

higher computation time than complex queries on smaller, less extensive datasets. Queries in such databases should especially benefit from the proposed optimizations.

4.6 JSONiq

As we solely express our queries in JSONiq, we provide the reader in this section with the need-to-know JSONiq syntax for the understanding of our queries. The complete JSONiq specification can be found in [6]. Here, we intentionally conflate the tuple streams produced by FLWOR operations with tree sequences, thus in this context make use of the JSONiq and XQuery terminology of "variable" as opposed to our use of "key" in more general JSON contexts.

- **How to input datasets?**
  There are two ways to input JSON data:

  1. **Internal input**: it is possible to create a collection of JSON documents in JSONiq itself. This is done by either copy-and-pasting an external collection or by simply writing, in an array, each JSON document. Generally, this will follow the syntax `let $collectionName := [{JSONdocument}, {JSONdocument}, {JSONdocument}, ...]`. A `let` binding assigns to a variable one or more values, in this case, a collection. To indicate the variable name in such a binding the name should start with `$` (note that `$` is not part of the actual variable name). To then assign a value, the variable name should be followed by `:=` and the concerned value, respectively. In case of the value being a collection, each JSON document is enclosed in curly brackets { }, where the syntax for object notation as described in Section 2 holds. All the JSON documents are comma-separated and enclosed by square brackets `[ ]`, which forms the actual collection.
     Example:
     A simple collection named *student* that contains one JSON document (as seen in Figure 3) can be created in JSONiq as follows:
     `let $student := [{"ID": 123456, "name":{"first": "Jane", "last": "Smith"}, "courses": ["EN", "SP", "MA", "GEO", "PHY", "SC", "CS", "PE"]}]`.
  2. **External input**: if data comes from an external source (e.g., locally saved file), it can be accessed in JSONiq by using the `collection()` function. The file-name or path-name of the collection is placed in-between the braces.
     Example:
     Loading in the *pokemon* collection from our first database in JSONiq is done through: `let $pokemon := collection("pokemon")`.

All of our queries start with `let $result :=`. We assign the output of the query to the variable `result` such that we can return the output of the query in a collection by `return [$result]`. It is possible in JSONiq to return the result of a query in other structures than a collection (e.g., a single document). However, due to the closedness of our JAL operators, we require all queries to return the query result in a collection.

Within our let-and-return clause, we allow for an arbitrary level of recursion of the following functions: `for`, `where`, and `order by`. Visualization for this syntax is provided in Figure 5. It can be seen that the query begins with a `let` statement, followed by an arbitrary level of recursion of the three aforementioned functions, and ends with two `return` statements. The first `return` describes how each JSON document looks in the query result, i.e., if it is desired to project only certain keys from the JSON document this can be requested in this first `return`. The second and final `return` statements collect all the JSON documents from the first `return` statement and aggregate them into a collection. This difference will be further clarified in the example at the end of this section.

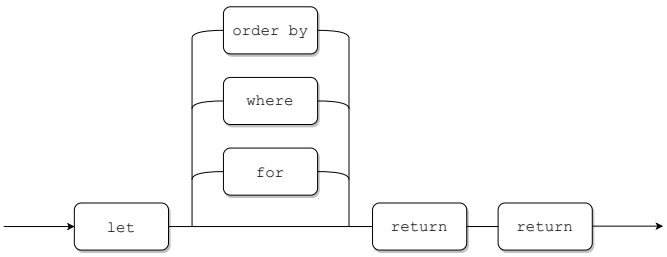


Fig. 5: Construction of a JSONiq query suitable for our implementation

- **`for` function**
  The `for` function loops over the documents of a collection. In its simplest form, this looks like `for $pokemon in $pokemon[]`, where `$pokemon[]` iterates over the collection with name `pokemon` and `$pokemon` is a JSON document (an iteration over the collection). A single `for` function can iterate over multiple collections, subsequently, by means of comma separation. For instance, such a `for` function over the bike-sharing data (Database 3) is modeled by: `for $station in $station[], $status in $status[]`. Similarly, this can also be expressed as subsequent `for` functions as expressed by: `for $station in $station[] for $status in $status[]`. Iterating over multiple collections is equivalent to a join-type operation in relational algebra.
- **`where` function**
  The `where` function specifies the conditions used for filtering the documents of a collection. It is equivalent to the selection operation familiar in relational algebra. A `where` function can consist of one or more conditions using comparison ($\{=, !=, >, >=, <, <=\}$) and logical operators (`and`, `or`, `not`). A condition starts with the key (or path as seen in Section 2.1) to a value in a JSON document, followed by a comparison operator, and ends with either an atomic value or another key to a value in a JSON document. If the key leads to a value of type array, and it is desired to check if a certain element is present in that array, JSONiq uses the `jn:members(key)` function (equivalent to using key[]) to iterate over all elements such that comparison is possible.
  Example 1:

This query asks to retrieve all the pokemon that weigh more than 10.0 kg and are smaller than 1.0 m from the collection *pokemon*. The `where` function in this JSONiq query is expressed as: `where $pokemon.weight > ‘‘10.0 kg’’ and $pokemon.height < ‘‘1.0 m’’`.
Example 2:
This query asks to retrieve all the pokemon which have as one of their weaknesses *Fire* from the collection *pokemon*. Note that the ‘weaknesses’ key leads to an array containing all of the weaknesses of the pokemon. The `where` function in this JSONiq query is expressed as: `where $pokemon.weaknesses[] = ‘‘Fire’’`.

- `order by` **function**
  The `order by` function is followed by a key (from a JSON document) and the word `ascending` or `descending`. If it is desired to order a collection on several keys, the `order by` function can be extended by key + `ascending` or `descending` statements separated by commas.
  Example:
  If it is desired to order the *pokemon* collection by name alphabetically, the `order by` function in JSONiq query is expressed as: `order by $pokemon.name ascending`.

Suppose we have the collections provided in the bike-sharing data from Section 3, and this query asks to retrieve all the station ids and region ids in ascending order of the stations that currently have more than 5 bikes available and that have 3 or less dockers available. The entire query expressed in JSONiq is given by:

```
let $stations := collection("station")
let $statuses := collection("status")
for $station in $stations,
    $status in $statuses
where $station.station_id = $status.station_id
  and $status.num_bikes_available > 5
  and $status.num_docks_available <= 3
order by $station.region_id ascending,
         $station.station_id ascending
return jn:project($station, ("region_id", "station_id"))
```

The first two lines are used to load and assign names to the input collections. In the next line, the actual query is stated, starting with a nested `for` function that indicates that we are dealing with some join-type operation. The `for` function is followed by a `where` function (containing multiple conditions) indicating that we are dealing with a join operation (follows from the first condition). In relational algebra, this boils down to a join of the two collections based on the three conditions. From the JSON documents that meet all the conditions, we solely want to project the region id and station id in ascending order, and return these documents in a collection. The output of this query is given by:

```
[{‘‘station_id’’: ‘‘hub_1267’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_299’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_309’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_321’’, ‘‘region_id’’: ‘‘region_80’’},
```

{‘‘station_id’’: ‘‘hub_3228’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_359’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_365’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_3793’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_5418’’, ‘‘region_id’’: ‘‘region_80’’},
{‘‘station_id’’: ‘‘hub_2456’’, ‘‘region_id’’: ‘‘region_81’’},
{‘‘station_id’’: ‘‘hub_2457’’, ‘‘region_id’’: ‘‘region_81’’},
{‘‘station_id’’: ‘‘hub_2459’’, ‘‘region_id’’: ‘‘region_81’’},
{‘‘station_id’’: ‘‘hub_2460’’, ‘‘region_id’’: ‘‘region_81’’},
{‘‘station_id’’: ‘‘hub_2529’’, ‘‘region_id’’: ‘‘region_81’’}].

Hence, a collection of JSON documents projecting only the desired keys “station_id” and “region_id”.

4.7 Implementation

For the implementation of our proposed algorithm, we used Jupyterlab, hence Python as our programming language. For the parsing of JSONiq queries, we use Rumble, a query execution engine for JSON running on top of Apache Spark, on a public server [41]. All code is processed on a computer with a 2.9 GHz Quad-Core Intel Core i5 processor and 16 GB 1600 MHz DDR3 memory. Our code and used queries are available at https://github.com/AnneJasmijnLangerak/JAL.

5 Results

This section describes the results that follow from our implementation. This section is divided into two parts. In the first part, we describe our use cases and give the theoretical and physical costs, as defined in Section 4.5, with and without optimization for each use case. In the second part, we provide an overview of the costs of all queries and make comparisons.

5.1 Use Cases

This section formulates our use cases. For each use case, we describe what the query looks like expressed in JSONiq, the translation to JAL displayed in query tree format, the query tree after applying our heuristic optimization algorithm, as well as the theoretical and physical costs of executing the query with and without optimization. This section is divided into five subsections, where each subsection describes two use cases belonging to one of the five databases that are presented in Section 3 and with physical costs determined using the code and system described in Section 4.7. We start with our smallest database in the first subsection, and end with our largest database in the last subsection.

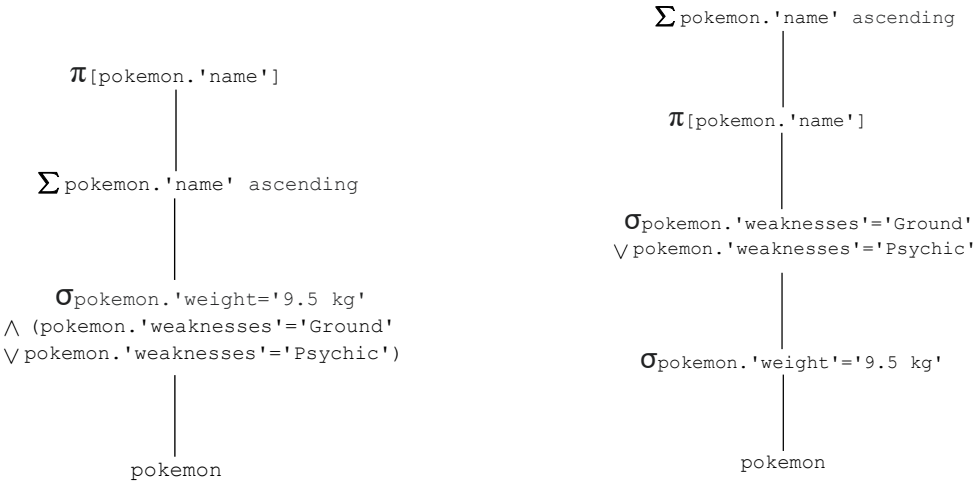
5.1.1 Pokémon (Database 1)

Use case 1: *Retrieve all pokemon names in alphabetical order that weigh 9.5 kg and*

have weaknesses Ground and/or Psychic.
For our first use case, the query expression in JSONiq takes on the following form:

```
let $pokemons := collection("pokemon")
for $pokemon in $pokemons
where $pokemon.weight = "9.5 kg"
  and ($pokemon.weaknesses[] = "Ground"
       or $pokemon.weaknesses[] = "Psychic")
order by $pokemon.name ascending
return { "name": $pokemon.name }
```

Translating this query into JAL results in the query tree displayed in Figure 6a. The theoretical cost of executing this query is 0, as there are no joins or Cartesian products present in the query. The physical cost of executing this query is 0.03 seconds. Optimizing this query results in the query tree in Figure 6b. It can be seen that the selection is decomposed into two selections. Following our heuristic optimization algorithm described in Section 4, the condition `$pokemon.weight = ‘‘9.5 kg’’` is executed first as this is the most restrictive condition. The other condition contains an `or` statement, hence the first condition takes precedence over the second condition. Again there is no theoretical cost. The physical cost of executing this optimized query is 0.01 seconds.



a Query tree before optimization

b Query tree after optimization

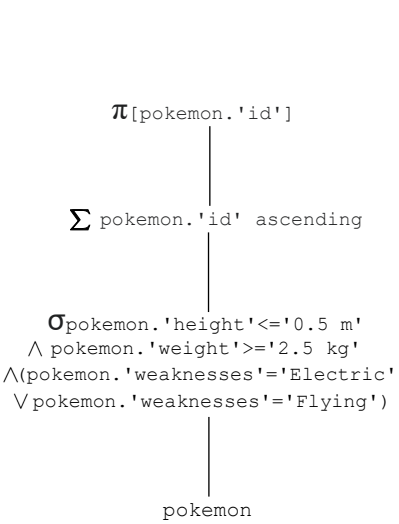
Fig. 6: Query tree for use case 1 before and after optimization

Use case 2: *Retrieve all pokemon ids in ascending order of pokemon that weigh 2.5 kg or more, have height 0.5 m or less and have weaknesses Electric and/or Flying.*
For our second use case, the query expression in JSONiq looks as follows:

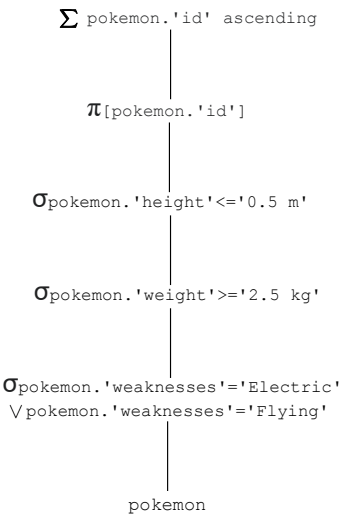
```
let $pokemons := collection("pokemon")
for $pokemon in $pokemons
```

```
where $pokemon.weight >= "2.5 kg"
  and $pokemon.height <= "0.5 m"
  and ($pokemon.weaknesses[] = "Electric"
       or $pokemon.weaknesses[] = "Flying")
order by $pokemon.id ascending
return { "id": $pokemon.id }
```

Translating this query into JAL results in the query tree displayed in Figure 7a. The theoretical cost of executing this query is 0, as there are no joins or Cartesian products present in the query. The physical cost of executing this query is 0.04 seconds. Optimizing this query results in the query tree in Figure 6b. It can be seen that the selection is decomposed into three selections. Following our heuristic optimization algorithm described in Section 4, the condition `$pokemon.weaknesses[]) = "Electric" or $pokemon.weaknesses[] = "Flying"` is executed first. Although it contains an `or` operator, it is still the most restrictive condition as all the comparison operators are equal operations. The other two conditions can be considered to be in the same category, as they both do not contain the equal operation and both do not contain the `or` operator. Hence, there is no way of knowing which restriction should be performed first. Again there is no theoretical cost. The physical cost of executing this optimized query is 0.03 seconds.



a Query tree before optimization



b Query tree after optimization

Fig. 7: Query tree for use case 2 before and after optimization

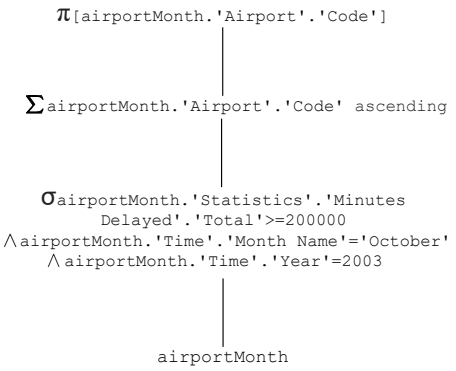
5.1.2 Monthly Airline Delays by Airport (Database 2)

Use case 3: *The retrieval of the airport codes in alphabetical order of all airports that*

experienced more than 200,000 minutes delay in October of 2003.
The query expression in JSONiq of our third case takes on the following form:

```
let $airportMonths := collection("airportMonth")
for $airportMonth in $airportMonths
where $airportMonth.Statistics."Minutes Delayed".Total >= 200000
  and $airportMonth.Time."Month Name" = "October"
  and $airportMonth.Time.Year = 2003
order by $airportMonth.Airport.Code ascending
return { "Code": $airportMonth.Airport.Code }
```

Translating this query into JAL results in the query tree displayed in Figure 8a. The theoretical cost of executing this query is 0, as there are no joins or Cartesian products present in the query. The physical cost of executing this query is 0.4 seconds. Optimizing this query results in the query tree in Figure 8b. It can be seen that the selection is decomposed into three selections. There are two conditions that both contain an equal operation and do not contain an `or` operator. Again, we can not know in real-time which condition should be performed first. Following our heuristic optimization algorithm, we do know that the selection with condition `$airportMonth.Statistics."Minutes Delayed".Total >= 200000` should be performed after the other two conditions, as this condition does not contain the equal operation. Again there is no theoretical cost. The physical cost of executing this optimized query is 0.3 seconds.



a Query tree before optimization

b Query tree after optimization

Fig. 8: Query tree for use case 3 before and after optimization

Use case 4: *The retrieval of the months and years in ascending order where airport LAX had more than 20 diverted flights while hosting less than 16,000 flights in that*

month.
The number of diverted flights are counted per month. The query expression in JSONiq takes on the following form:

```
let $airportMonths := collection("airportMonth")
for $airportMonth in $airportMonths
where $airportMonth.Airport.Code = "LAX"
  and $airportMonth.Statistics.Flights.Total < 16000
  and $airportMonth.Statistics.Flights.Diverted > 20
order by $airportMonth.Time.Year ascending,
         $airportMonth.Time."Month Name" ascending
return {
     "Month Name": $airportMonth.Time."Month Name",
     "Year": $airportMonth.Time.Year
}
```

Translating this query into JAL results in the query tree displayed in Figure 9a. The theoretical cost of executing this query is 0, as there are no joins or Cartesian products present in the query. The physical cost of executing this query is 0.4 seconds. Optimizing this query results in the query tree in Figure 9b. It can be seen that the selection is decomposed into three selections. Following our heuristic optimization algorithm described in Section 4, the condition `$airportMonth.Airport.Code="LAX"` is executed first. The other two conditions can be considered to be in the same category, as they both do not contain the equal operation and both do not contain the `or` operator. Hence, there is no way of knowing which restriction should be performed first. Again there is no theoretical cost. The physical cost of executing this optimized query is 0.3 seconds.

a Query tree before optimization

b Query tree after optimization

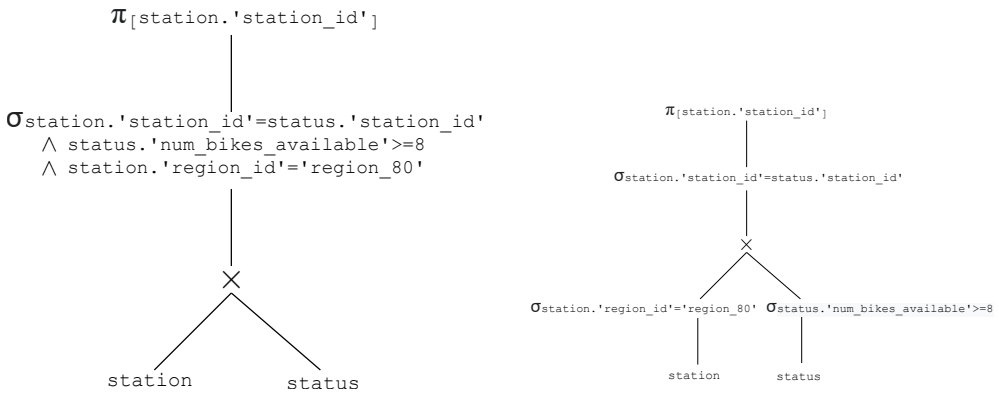
Fig. 9: Query tree for use case 4 before and after optimization

5.1.3 Bike Sharing at Stations (Database 3)

Use case 5: *The retrieval of the station ids of all stations in the region with id 80 that have 8 or more bikes available at the station.*
The query expression in JSONiq takes on the following form:

```
let $stations := collection("station")
let $statuses := collection("status")
for $station in $stations, $status in $statuses
where $station.station_id = $status.station_id
  and $status.num_bikes_available >= 8
  and $station.region_id = "region_80"
return { "station_id": $station."station_id" }
```

Translating this query into JAL results in the query tree displayed in Figure 10a. The theoretical cost of executing this query can be computed by $|station| \times |status|$, hence $95 \times 95 = 9,025$. The theoretical cost of this query tree is thus 9,025 elements. The physical cost of executing this query is 1.8 seconds. Optimizing this query results in the query tree in Figure 10b. It can be seen that the selection is decomposed into three selections. Two of those selections contain a condition that restricts the selection to a single collection. Hence, they are to be performed before the selection with the condition `$station.station_id = $status.station_id`. As the two conditions both make a selection over a different collection, precedence of one over the other is not relevant. The collection *station* after its corresponding selection yields an intermediate output with a quantitative dimension of 59. The collection *status* after its corresponding selection yields an intermediate output with a quantitative dimension of 14. The Cartesian product of these results leaves us with a theoretical cost of $59 \times 14 = 826$ elements. The physical cost of executing this optimized query is 0.1 seconds.



a Query tree before optimization

b Query tree after optimization

Fig. 10: Query tree for use case 5 before and after optimization

Use case 6: *The retrieval of the station ids and region ids of all stations where the number of available bikes is larger than number of available dockers except for stations in the region 80.*
The query expression in JSONiq takes on the following form:

```
let $stations := collection("station")
let $statuses := collection("status")
for $station in $stations, $status in $statuses
where $station.station_id = $status.station_id
  and $status.num_bikes_available > $status.num_docks_available
  and $station.region_id != "region_80"
return jn:project($station, ("station_id", "region_id"))
```

Translating this query into JAL results in the query tree displayed in Figure 11a. The theoretical cost of executing this query can be computed by $|station| \times |status|$, hence $95 \times 95 = 9,025$. The theoretical cost of this query tree is thus 9,025 elements. The physical cost of executing this query is 2.4 seconds. Optimizing this query results in the query tree in Figure 11b. It can be seen that the selection operator is decomposed into three selections. The selection with the condition `$station.region_id != ‘‘region_80’’` is pushed down further in the tree than the two other conditions. This is because this condition is restricted to a single collection, while the other two selections contain a condition involving both collections. The next selection that is executed is the one with the condition `$station.station_id = $status.station_id` as it contains the equal operation. Last, the selection with condition `$status.num_bikes_available > $status.num_docks_available` is executed, as it does not contain an equal operation and involves two collections. The collection *station* after its corresponding selection yields an intermediate output with a quantitative dimension of 36. The Cartesian product of this result with the collection *status* leaves us with a theoretical cost of $36 \times 95 = 3,420$ elements. The physical cost of executing this optimized query is 0.6 seconds.

a Query tree before optimization

b Query tree after optimization

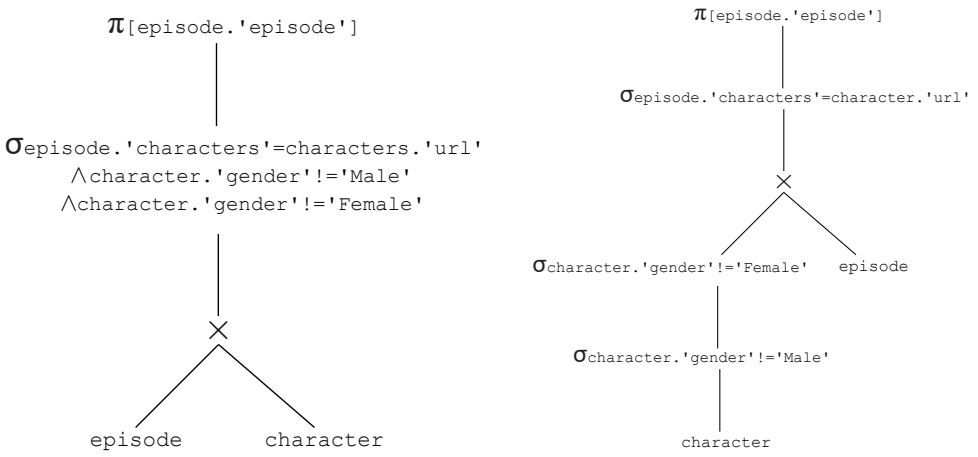
Fig. 11: Query tree for use case 6 before and after optimization

5.1.4 Rick and Morty TV-Show (Database 4)

Use case 7: *The retrieval of the episode codes where at least one of the characters from the episode is neither female nor male.*
An example of episode code is S01E01, meaning season 1 episode 1. The query expression in JSONiq takes on the following form:

```
let $episodes := collection("episode")
let $characters := collection("character")
for $episode in $episodes
for $character in $characters
where $episode.characters.url = $character.url
where $character.gender != "Male"
  and $character.gender != "Female"
return { "episode": $episode.episode }
```

Note that if an episode has more than one character that is neither male nor female, the episode will occur more than once in the output collection. One can remove these duplicates by using the *distinct* operator presented in Section 4. Translating this query into JAL results in the query tree displayed in Figure 12a. The theoretical cost of executing this query can be computed by $|episode| \times |character|$, hence $41 \times 671 = 27,511$. The theoretical cost of this query tree is thus 27,511 elements. The physical cost of executing this query is 6.1 seconds. Optimizing this query results in the query tree in Figure 12b. It can be seen that the selection operator is decomposed into three selections. Two of those selections involve a single collection (namely *character*) and both of their conditions do not contain `or` operators or equal operations. Hence, we do not know in real-time which selection should be performed first. What does follow from our optimization algorithm is that the selection with condition `$episode.characters.url = $character.url` should be executed after the other two, as it involves two collections. The collection *character* after the two single selections yields an intermediate output with a quantitative dimension of 61. The Cartesian product of this intermediate result with the collection *episode* costs $61 \times 41 = 2,501$. Hence, the theoretical cost of the optimized query is 2,501 elements. The physical cost of executing this optimized query is 0.5 seconds.



a Query tree before optimization

b Query tree after optimization

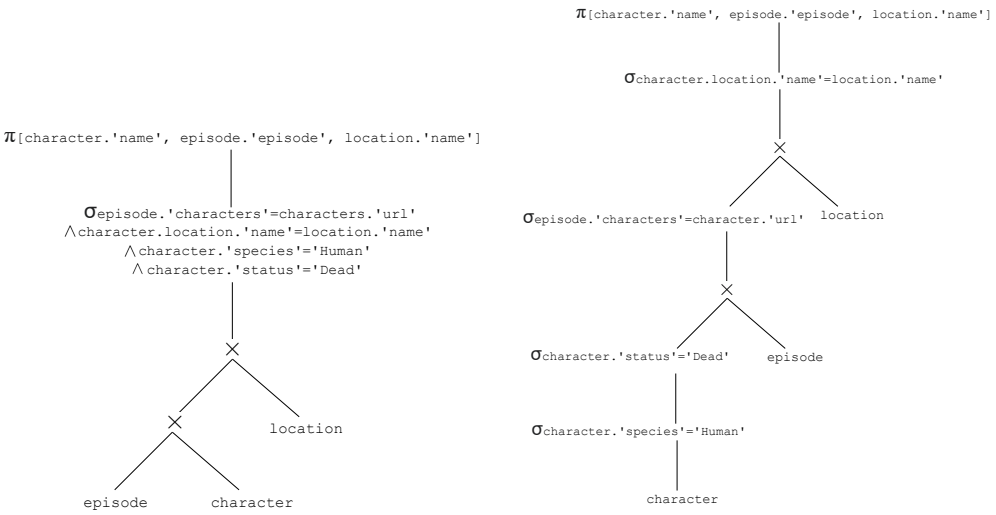
Fig. 12: Query tree for use case 7 before and after optimization

Use case 8: *The retrieval of the episode code, character name, and location of the episodes where at least one of the characters from that episode is human and the character's current status is dead.*
The query expressed in JSONiq takes on the following form:

```
let $episodes := collection("episode")
let $characters := collection("character")
let $locations := collection("location")
for $episode in $episodes, $character in $characters
where $episode.characters = $character.url
for $location in $locations
where $character.location.name = $location.name
where $character.species = "Human"
  and $character.status = "Dead"
return {
     "name": $character.name,
     "episode": $episode.episode,
     "location": $location.name
}
```

Again, if such a character occurs in more than one episode, or an episode contains more than one of such characters, both will occur more than once in the output. One can remove these duplicates by using the *distinct* operator presented in Section 4. Translating this query into JAL results in the query tree displayed in Figure 13a. The theoretical cost of executing this query can be computed by $|episode| \times |character| \times |location|$, hence $41 \times 671 \times 108 = 2,971,188$. The theoretical cost of this query is thus $2,971,188$ elements. The physical cost of executing this query is $3,516.1$ seconds

(about 59 minutes). Optimizing this query results in the query tree displayed in Figure 13b. It can be seen that the selection operator is decomposed into four selections. Two of which solely involve the *character* collection and both conditions strictly contain the equal operation. There is no way of knowing in real-time which of these two selections should be performed first. The remaining two selections can both be placed within the same category: both involve two collections, both involve the equal operations, and both do not contain any `or` operators. Again, we do not know which selection will result in the smallest (intermediate) output and thus we do not know which of the two selections should be executed first. The two selections over the collection *character* result in an intermediate output of size 80. The Cartesian product of this result with the collection *episode* yields $80 \times 41 = 3,280$ elements. The selection over this output gives us an intermediate result with size 92. The Cartesian product of this result with the collection *location* results in $92 \times 108 = 9,936$ elements. Hence, the theoretical cost of this optimized query is $3,280 + 9,936 = 13,216$ elements. The physical cost of executing this query is 2.1 seconds.



a Query tree before optimization

b Query tree after optimization

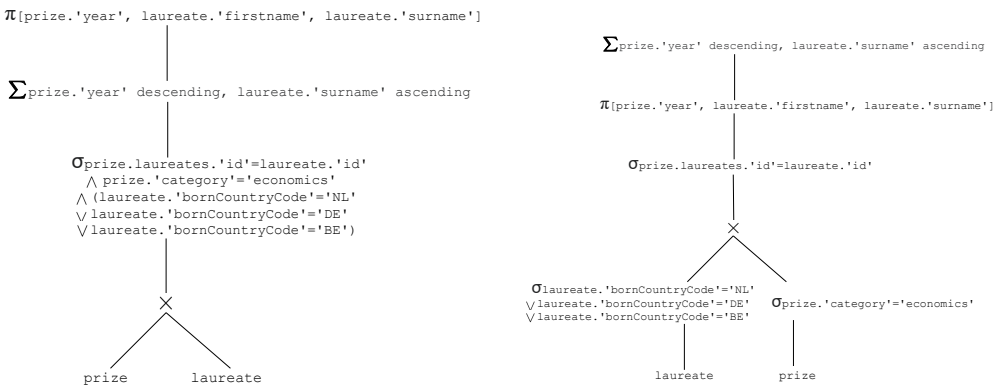
Fig. 13: Query tree for use case 8 before and after optimization

5.1.5 Nobel Prizes and Winners (Database 5)

Use case 9: *The retrieval of the year the Nobel Prize was received, and the first name and surname of all Dutch, German, and Belgian Nobel Prize Laureates that won a Nobel Prize in the category economics, sorting the result in descending order by year and ascending order by surname.*
The query expressed in JSONiq takes on the following form:

```
let $prizes := collection("prize")
let $laureates := collection("laureate")
for $prize in $prizes
for $laureate in $laureates
where $prize.laureates.id = $laureate.id
  and $prize.category = "economics"
where $laureate.bornCountryCode = ("NL", "DE", "BE")
order by $prize.year descending,
         $laureate.surname ascending
return {
     "year": $prize.year,
     "firstname": $laureate.firstname,
     "surname": $laureate.surname
}
```

Translating this query into JAL results in the query tree shown in Figure 14a. The theoretical cost of executing this query can be computed as $|prize| \times |laureate|$, hence $652 \times 955 = 622,660$. The theoretical cost of this query is thus 622, 660 elements. The physical cost of executing this query is 522.3 seconds. Optimizing this query tree results in the query tree provided in Figure 14b. It can be seen that the selection operator is decomposed into three selections. There are two selections with a condition involving a single collection. Both selections involve a different collection, hence their relative order of execution is not relevant. The condition in the third selection involves two collections, hence this selection should be executed after the other two selections. The theoretical cost of this optimized query is computed as follows: the collection *prize* contains 52 nodes after the selection. Similarly, the collection *laureate* contains 109 nodes after the selection is performed. The Cartesian product of both results in $52 \times 109 = 5,668$ elements. Hence, the theoretical cost of executing the optimized query is 5, 668. The physical cost is 1.0 seconds.



a Query tree before optimization

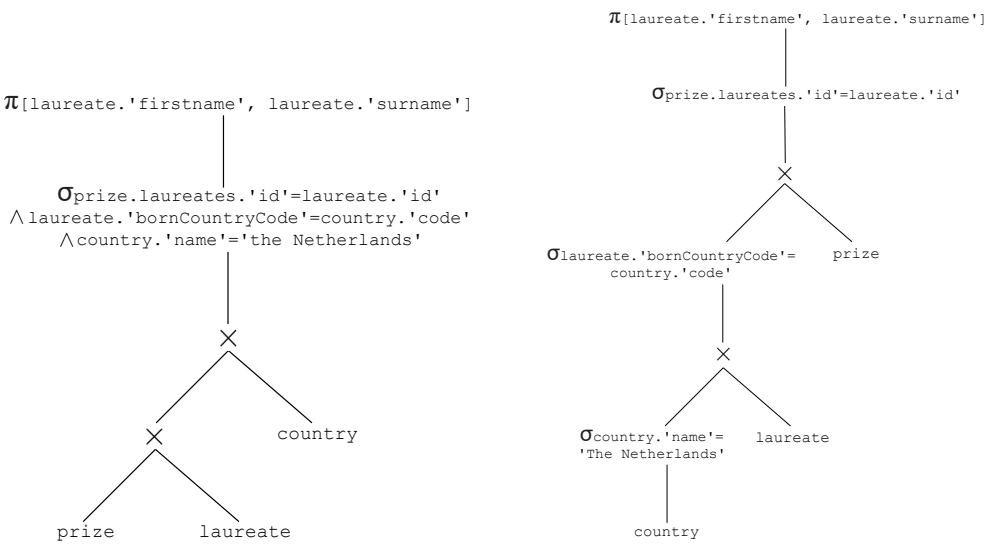
b Query tree after optimization

Fig. 14: Query tree for use case 9 before and after optimization

Use case 10: *The retrieval of the first name and surname of all Dutch Nobel Prize Laureates.*
For our final use case, the query expressed in JSONiq takes on the following form:

```
let $prizes := collection("prize")
let $laureates := collection("laureate")
let $countries := collection("country")
for $prize in $prizes
for $laureate in $laureates
where $prize.laureates.id = $laureate.id
for $country in $countries
where $laureate.bornCountryCode = $country.code
  and $country.name = "The Netherlands"
return jn:project($laureate, ("firstname", "surname"))
```

Translating this query into JAL results in the query tree displayed in Figure 15a. The theoretical cost of executing this query can be computed by $|prize| \times |laureate| \times |country|$, hence $652 \times 955 \times 135 = 84,059,100$. The theoretical cost of this query tree is thus $84,059,100$ elements. We cannot compute the physical cost of this query due to the high running time executing this query requires. After four hours our kernel shut down, thus leaving us without a physical cost. Optimizing this query tree results in the query tree provided in Figure 15b.



a Query tree before optimization

b Query tree after optimization

Fig. 15: Query tree for use case 10 before and after optimization

It can be seen that the selection operator is decomposed into three selections. There is one selection with a condition involving a single collection. Therefore, this selection

is pushed further down the query tree than the other two selections. The conditions in these two selections can be placed within the same category: both involve two collections, both contain the equal operation, and both do not contain `or` operators. There is no way of knowing in real-time which of the two should take precedence over the other. The theoretical cost of this optimized tree is computed as follows: the selection over the collection *country* yields an output containing a single node. The Cartesian product with the *laureate* collection results in $1 \times 955 = 955$ elements. Performing the next selection over this intermediate output results in a collection containing 18 nodes. The Cartesian product of this new intermediate output with the collection *prize* results in $18 \times 652 = 11,736$. Next we have a selection and a projection of the result. Hence, the theoretical cost of the query after optimization is $955 + 11,736 = 12,691$ elements. The physical cost of executing the optimized query is 1.8 seconds.

5.2 Overview

An overview of both the theoretical as well as the physical costs (as determined using the system details of Section 4.7) for all use cases is given in Table 6. More precisely, Table 6 shows the theoretical costs, which are determined by the cardinality of the joins, hence qualitative results. It also shows the physical costs (in seconds), which are determined by the times resulting from running the queries, hence quantitative results. Our smallest database (*D1*) covers the data for one collection that consists of 151 JSON documents and our largest database (*D5*) covers data for three collections of sizes 652, 955, and 135 JSON documents. For all use cases where a join-type operation was required (use cases 5 to 10), the physical cost as well as the theoretical cost decreased significantly after optimization was applied. For use cases performing a query without a join-type operation, only an improvement in physical cost is observed. As theoretical costs depend on the presence of join-type operations, it follows that there is no theoretical cost for such queries.

The larger the size(s) (i.e., number of JSON documents) of the collection(s), the more a significant difference can be observed between the costs of executing the query with and without optimization. For instance, when comparing use case 5 (dataset 3 (D3) covering two collections both with a size of 95 JSON documents) with use case 9 (dataset 5 (D5)), the difference of physical cost after optimization is substantially larger. The physical cost of use case 5 decreases from 1.8 seconds to 0.1 seconds, whereas the physical cost of use case 9 decreases from 522.3 seconds to 1.0 seconds.

		Without Optimization Algorithm		*With Optimization Algorithm*	
		Theoretical Cost (# pairs)	Physical Cost (in s)	Theoretical Cost (# pairs)	Physical Cost (in s)
D1	Use case 1	0	0.03	0	0.01
	Use case 2	0	0.04	0	0.03
D2	Use case 3	0	0.40	0	0.30
	Use case 4	0	0.40	0	0.30
D3	Use case 5	9,025	1.80	826	0.10
	Use case 6	9,025	2.40	3,420	0.60
D4	Use case 7	27,511	6.10	2,501	0.50
	Use case 8	2,971,188	3,516.10	13,216	2.10
D5	Use case 9	622,660	522.30	5,668	1.00
	Use case 10	84,059,100	-	12,691	1.80

Table 6: The theoretical and physical costs of executing all the 10 use cases with and without optimization

6 Conclusion

While JSON has risen in usage compared to other data formats in large database applications, JSON querying approaches have not enjoyed the same algebraic optimization that has been developed for queries of other data formats like XML. In this paper we have defined a tree-based JSON data model, a JSON algebra with algebraic operators, algebraic operator equivalence rules, as well as a heuristic optimization algorithm for the purpose of optimizing JSON queries. Through this we have provided an approach for the logical optimization of JSON queries, extending approaches developed in relational algebra and XML contexts into a JSON context. We find that JSON queries can indeed be optimized by employing our heuristic optimization algorithm through translation into an algebraic form by converting the relevant collection(s) into our proposed JSON Data Model and making use of our algebraic equivalence expressions. In particular, we tested in 10 use cases based on 5 databases with varying structures and levels of query cost. There are considerable reductions in both theoretical and physical costs in execution when queries are optimized compared to when they are not. The reduction in cost becomes more considerable when the original query is costlier. Even in cases where there are no theoretical costs associated with either optimized or unoptimized queries, our approach still shows a reduction in the physical cost of execution.

We have several suggestions for directions of future research. One may have noticed that our use cases focus solely on projections, selections, join-type operations, and sorting. Our reasoning for this is that we expect queries containing such operations to best showcase our algorithm's impact on the execution costs of JSON queries. However, in the implementation of our algorithm, we did create and test all other unary and binary operators we defined in Section 4.2. There may be value in the investigation of the efficiency gain (or lack thereof) that can be attained by our algorithm for queries using (a combination) of those operators. Furthermore, improvements can be made regarding our implementation, such as the addition of a proper parser for JSON

data, as we now assume that the (input) data has a valid structure without performing validation. Additionally, extending the JSONiq engine with the implementation of our algorithm, for the comparison of query execution with and without our optimization algorithm, may be a relevant aim for the continuation of this work. Also of interest for future work are the consequences of extending our approach into a distributed context and analysis of the feasibility of such an implementation compared to single-machine environments. Further research into the combination of these logical optimization approaches with some form of physical optimization may be warranted, with possible additional efficiency gains available when both are used in tandem. Last, we would like to extend our approach currently focusing on syntactic querying to semantic querying inspired by recent work proposed for semi-structured data [42, 43].

Declarations

Ethical Approval

Not applicable.

Competing interests

The authors declare that they have no conflict of interest.

Authors' contributions

Anne Jasmijn Langerak did the implementation. Flavius Flasincar and Jasmijn Klinkhamer review the manuscript. All authors wrote the manuscript.

Funding

Not applicable.

Availability of data and materials

The code and the dataset are available at https://github.com/AnneJasmijnLangerak/JAL.

References

[1] Florescu, D., Fourny, G.: JSONiq: The history of a query language. IEEE Internet Computing **17**(5), 86–90 (2013)

[2] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML) version 1.0, W3C recommendation November 26th 2008. http://www.w3.org/TR/REC-xml (2008)

[3] Crockford, D.: Introducing JSON. https://www.json.org/json-en.html. Accessed: 2025-01-21 (2023)

[4] Ullman, J.D.: Principles of Database and Knowledge-Base Systems vol. 1&2. Computer Science Press (1989)

[5] Frasincar, F., Houben, G., Pau, C.: XAL: An algebra for XML query optimization. In: Thirteenth Australasian Database Conference (ADC 2002). CRPIT, vol. 5. Australian Computer Society (2002)

[6] Robie, J., Fourny, G., Brantner, M., Florescu, D., Westmann, T., Zaharioudakis, M.: JSONiq. https://www.jsoniq.org. Accessed: 2025-01-21 (2023)

[7] Bourhis, P., Reutter, J.L., Suárez, F., Vrgoč, D.: JSON: Data model, query languages and schema specification, pp. 123–135. The 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2017), ACM (2017)

[8] Chortaras, A., Stamou, G.: D2RML: Integrating heterogeneous data and web services into custom RDF graphs. In: Workshop of Linked Data on the Web 2018 (LDOW 2018). CEUR Workshop Proceedings, vol. 2073. CEUR-WS.org (2018)

[9] Hidders, J., Paredaens, J., Bussche, J.: J-Logic: Logical foundations for JSON querying. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2017), pp. 137–149. ACM (2017)

[10] Goessner, S.: JSONPath - XPath for JSON. https://goessner.net/articles/JsonPath/. Accessed: 2025-01-21 (2007)

[11] Beyer, K.S., Ercegovac, V., Gemulla, R., Balmin, A., Eltabakh, M., Kanne, C.-C., Ozcan, F., Shekita, E.J.: JAQL: A scripting language for large scale semistructured data analysis. The VLDB Endowment **4**(12), 1272–1283 (2011)

[12] Zyp, K.: JSONQuery: Data querying beyond JSONPath. https://www.sitepen.com/blog/jsonquery-data-querying-beyond-jsonpath. Accessed: 2025-01-21 (2008)

[13] Saryerwinnie, J.: JMESPath: JMESPath is a query language for JSON. https://jmespath.org. Accessed: 2025-01-21 (2015)

[14] Kalbarczyk, A.: The agile NoSQL query language for semi-structured data. https://pypi.org/project/objectpath/t/. Accessed: 2025-01-21 (2018)

[15] JSONata.org: JSONata. http://docs.jsonata.org/overview.html. Accessed: 2025-01-21 (2021)

[16] Ong, K.W., Papakonstantinou, Y., Vernoux, R.: The SQL++ query language: Configurable, unifying and semi-structured. arXiv preprint arXiv:1405.3631 (2014)

[17] Capelli, S., Fosci, P., Marini, F., Psaila, G.: J-CO-QL: A flexible query language for complex geographical analysis of heterogeneous geo-tagged JSON data sets. University of Bergamo, Dalmine, Italy (2017)

[18] Clark, J., DeRose, S.: XML path language (XPath), W3C recommendation 16 November 1999. https://www.w3.org/TR/1999/REC-xpath-19991116/ (2016)

[19] Bourhis, P., Reutter, J.L., Vrgoč, D.: JSON: Data model and query languages. Information Systems **89**, 101478 (2020)

[20] Google: JAQL. https://code.google.com/archive/p/jaql/. Accessed: 2025-01-21 (2023)

[21] Lee, J., Anjos, E., Satti, S.R.: SJSON: A succinct representation for JSON documents. Information Systems **97**, 101686 (2021)

[22] Robie, J., Dyck, M., Spiegel, J.: XQuery 3.1: An XML query language, W3C recommendation 21 March 2017. https://www.w3.org/TR/xquery-31/ (2017)

[23] Fourny, G.: JSONiq, the SQL of NoSQL. https://www.jsoniq.org/docs/Introduction_to_JSONiq/xml_tmp/out/pdf/Introduction_to_JSONiq.pdf (2013)

[24] Landelle, S.: Yet another JSON query language: Introducing JMESPath support. https://gatling.io/2019/07/31/introducing-jmespath-support/. Accessed: 2025-01-21 (2019)

[25] Psaila, G., Fosci, P.: J-CO: A platform-independent framework for managing geo-referenced JSON data sets. Electronics **10**(5), 621 (2021)

[26] Grigorev, A.: Logical query plan optimization. http://mlwiki.org/index.php/Logical_Query_Plan_Optimization. Accessed: 2025-01-21 (2014)

[27] Hogenboom, A., Niewenhuijse, E., Hogenboom, F., Frasincar, F.: RCQ-ACS: RDF chain query optimization using an ant colony system. In: 2012 IEEE/WIC/ACM International Conferences on Web Intelligence (WI 2012), pp. 74–81. IEEE Computer Society (2012)

[28] Grigorev, A.: Physical operators (databases). http://mlwiki.org/index.php/Physical_Operators_(databases). Accessed: 2025-01-21 (2013)

[29] PostgreSQL: JSON functions and operators. https://www.postgresql.org/docs/current/functions-json.html. Accessed: 2025-01-21 (2023)

[30] IBM: JSON queries. https://www.ibm.com/docs/en/db2/11.5?topic=planning-json-queries. Accessed: 2025-01-21 (2023)

[31] Truică, C.-O., Darmont, J., Boicea, A., Rădulescu, F.: Benchmarking top-k keyword and top-k document processing with T2K2 and T2K2D2. Future Generation

Computer Systems **85**, 60–75 (2018)

[32] Durner, D., Leis, V., Neumann, T.: JSON tiles: Fast analytics on semi-structured data. In: 2021 International Conference on Management of Data (SIGMOD 2021), pp. 445–458. ACM (2021)

[33] Andrei, C., Florescu, D., Fourny, G., Robie, J., Velikhov, P.: JSound: A formal yet very simple JSON schema language. http://www.jsound-spec.org/. Accessed: 2025-01-21 (2025)

[34] Dorfman, J.: Awesome JSON datasets. https://github.com/jdorfman/awesome-json-datasets. Accessed: 2025-01-21 (2023)

[35] Tempe, C.: GRID bikeshare data. https://data.world/city-of-tempe/40b49ebc-ff11-43ed-b814-b5137fc53b4c/workspace/project-summary?agentid=city-of-tempe&datasetid=40b49ebc-ff11-43ed-b814-b5137fc53b4c. Accessed: 2025-01-21 (2023)

[36] Hamadou, H.B., Jedidi, F.G., Péninou, A., Teste, O.: Towards schema-independent querying on document data stores. In: 20th International Workshop On Design, Optimization, Languages and Analytical Processing of Big Data (DOLAP 2018). CEUR Workshop Proceedings, pp. 1–10. CEUR-WS.org (2018)

[37] Peyton Jones, S.L.: The Implementation of Functional Programming Languages. Prentice-hall International Series in Computer Science. Prentice-Hall (1987)

[38] Moggi, E.: Computational lambda-calculus and monads. In: Fourth Annual Symposium on Logic in Computer Science (LICS 1989), pp. 14–23. IEEE Press (1989)

[39] Elmasri, R., Navathe, S.: Fundamentals of Database Systems, 7th edn. Pearson (2017)

[40] Tutorial Points: Query optimization in centralized systems. https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_query_optimization_centralized_systems.html. Accessed: 2025-01-21 (2023)

[41] Müller, I., Fourny, G., Irimescu, S., Cikis, C.B., Alonso, G.: Rumble: Data independence for large messy data sets. Proceedings of the VLDB Endowment **14**(4), 498–506 (2020)

[42] Tekli, J., Tekli, G., Chbeir, R.: Combining offline and on-the-fly disambiguation to perform semantic-aware XML querying. Computer Science and Information Systems **20**(1), 423–457 (2023)

[43] Tekli, J., Chbeir, R., Traina, A.J., Traina Jr, C.: Semindex+: A semantic indexing scheme for structured, unstructured, and partly structured data.

Appendix A RegEx Grammar

To elaborate further on the RegEx lexer grammar defined in Table 5, JSONiq queries follow a certain structure. Because of this structure, we can identify the tasks the query consists of. This is done by the lexer rules:

- The first rule *VARIABLE ASSIGNMENT* is required to extract the part in which JSONiq stores its query result. This always starts with the word *let* followed by a dollar sign *$* (to announce a name) and some characters representing the name of the result, and ending with *:=*. Note that *let*, *$*, and *:=* are thus JSONiq syntax. This rule is written in RegEx. Therefore, a \ must added before *$* to be able to capture this symbol. As the name of the query result can be of irregular length, RegEx uses **.+** to allow for one or more characters of input in that position. The variable assignment is complete once *:=* has been found. RegEx can recognize this as the end of this rule, by placing **( )** around *:=*.
- The second rule *PROJECTION* is used for retrieving the (sub)set of columns used for the output. JSONiq uses *for* to select one or more columns. As columns have names, *for* is followed by a *$*. RegEx then captures the name using **.+**, allowing for the name to contain one or more characters. Following JSONiq syntax, the column name belongs to some table. This can be recognized by the word *in*, followed by the name of the table. Again, RegEx matches this name by using **.+**. The regular expression **?(?=for—where—return—for—let—order by)** is a positive lookahead assertion. It specifies a condition that can be optionally true for a match to occur (this can be recognized by **?** before the parenthesis), but it doesn't include the matched keyword in the overall match.
- The third rule *CONDITION* is used for filtering. Filtering in JSONiq is done by using *where*, followed by the column name and the actual filtering condition. The column name starts with a *$*, following JSONiq syntax. To capture the actual filter condition, RegEx uses **.***, which means zero or more characters. The regular expression **(?=for—return—let—order by—where)** is a positive lookahead assertion. It specifies a condition that can be optionally true for a match to occur (this can be recognized by **?** before the parenthesis), but it doesn't include the matched keyword in the overall match.
- The fourth rule *RETURN* covers the formatting of the output of query. *return* in JSONiq indicates what the format of the result will look like. It is followed by one or more characters defining the format. RegEx captures this by using **.+**. This rule closes with the positive lookahead assertion **?(?=return)**. It specifies that the rule can optionally end with another return. If such a match is found, it is not included in the overall match.
- The fifth rule *SORT* can be used to extract the ordering of the output of the query. JSONiq allows for the sorting of output of the query by using *order by*, followed optionally by the name of the column on which the output should be ordered. RegEx captures this by means of **.***, which means zero or more characters. The regular expression **(?=return—for—let—order by)** is a positive lookahead assertion. It

specifies a condition that can be optionally true for a match to occur (this can be recognized by **?** before the parenthesis), but it doesn't include the matched keyword in the overall match.