Algebraic Mapping Operators for Knowledge Graph Generation

Sitt Min Oo [a,*], Ben De Meester [a], Ruben Taelman [a] and Pieter Colpaert [a]

[a] *Department of Engineering and Architecture, University of Ghent – imec, Ghent, Belgium*
E-mails: x.sittminoo@ugent.be, ben.demeester@ugent.be, ruben.taelman@ugent.be, pieter.colpaert@ugent.be

Abstract. Recent advancements in declarative knowledge graph generation have led to the development of multiple mapping languages, their various versions, and different mapping engines that can interpret these languages and execute the mapping process. The field has progressed to the extent that current studies are now more focused on optimizing the knowledge graph generation process. Although different mapping engines share the common functionality of generating knowledge graphs from heterogeneous data sources, sharing the various optimization techniques and features of these engines remains challenging due to the lack of formal operational semantics for the general mapping processes. A set of algebraic mapping operators can provide the necessary operational semantics for general mapping processes, establish a theoretical foundation for mapping languages, and facilitate the introduction and evaluation of a compliant implementation, that is capable of interpreting and executing multiple mapping languages. In this paper, we propose such an algebra based on the SPARQL algebra. This allows us to maximally reuse established definitions, and further bridge the world of knowledge graph generation with query engines. To evaluate that our work is not limited to a single specific mapping language, we translated mapping languages ShExML and RML to our mapping plan composed of algebraic mapping operators. The results of our completeness evaluation shows that our algebraic operators cover the operational semantics of RML and partially for ShExML. To fully cover ShExML, further analysis into ShExML's concise operational semantics is needed (e.g. for joining data from two input sources). For performance evaluation, our proof-of-concept algebraic mapping engine has a consistent and low memory usage across the different workloads, and achieved second place in the Knowledge Graph Construction Workshop's performance challenge. Algebraic mapping operators decouple mapping engines from the mapping languages, enabling multilingual mapping engines. Furthermore, the mapping plan can incorporate optimization techniques as a separate process from the mapping itself, allowing us to benefit from state-of-the-art mapping process optimizations. The proposed set of algebraic mapping operators will lay the foundation for future studies on the theoretical analysis of complexity and expressiveness of mapping languages, and will provide consistency in the execution semantics of mapping engines. Furthermore, the alignment of our algebra with SPARQL will enable further research into advanced methods such as virtualization, enabling heterogeneous data querying.

Keywords: Mapping Algebra, Semantic Web, Mapping Language, Knowledge Graph Generation

1. Introduction

There exist several *use-case-agnostic and declarative* mapping languages [1–7] to generate a Knowledge Graph (KG) using the Resource Description Framework (RDF) [8]. *Mapping rules* – described using such mapping language – specify the *mapping process*: how to generate a KG from existing semi-structured data [8]. These mapping languages' increasing popularity and importance is signified by the establishment of the W3C Knowledge

*Corresponding author. E-mail: x.sittminoo@ugent.be.

Graph Construction Community Group, and a diverse ecosystem of mapping engines based on these mapping languages [8]. For example, several mapping engines based on the RDF Mapping Language (RML) [7] have been implemented [3, 9–13], and there are plans to establish a RML W3C Working Group in the future [14].

However, these mapping languages typically provide no formal description of the *operational semantics* through *mapping plans*: the formally defined steps a mapping process should follow to generate a KG based on mapping rules. This prevents the static analysis of the mapping languages to prove the correctness of the mapping process, and analyse the expressiveness and the complexity of the mapping languages. Attempts have been made to formalize these mapping languages, however, these formalizations are mostly used in the context of proving the correctness of an optimization technique [15] or restricted to a particular mapping language [16, 17]. To the best of our knowledge, there is currently no research on the theoretical foundations, independent of specific mapping languages, that is applicable to multiple mapping languages.

Hence, different mapping engine implementations typically *individually* infer the operational semantics of the mapping plan based on the *syntax* of the mapping language. As a result, the operational semantics of the mapping engines are different even when using the same mapping rules. These individual inferences of the operational semantics lead to a slow-down in mapping engine development, with repetitive implementations of the same operational semantics in different flavours. Additionally, performance optimizations are scattered across different engine implementations; all incompatible with each other due to the aforementioned individual inferences of the operational semantics.

In this work, we provide such a mapping language-independent theoretical foundation by defining a set of algebraic operators for the mapping process called *mapping algebra*, adapting the algebra [18] of SPARQL, an RDF data query language. These algebraic mapping operators can be composed together to form a mapping plan. We provide a proof-of-concept implementation on par – both in performance as in functionality – with existing mapping engine implementations. This shows how to translate mapping rules from multiple mapping languages into a mapping plan, and thereby providing operational semantics for mapping languages.

This approach can lead to more aligned mapping engines, and allows proving correctness across different mapping languages. Adapting the SPARQL algebra [18] to formulate the mapping algebra allows us to maximally reuse established definitions.

The outline of this paper is as follows: in Section 2, we explore the state-of-the-art on mapping languages and their operational semantics. In Section 3, we discuss our methodology and its rationale. In Section 4, we provide the formal terms and definitions of the set of algebraic mapping operators used to construct the mapping plan. In Section 5, we introduce a reference implementation that consists of an algebraic mapping translator and a proof of concept engine. In Section 6, we provide an overview of our evaluation methodology and results, to show the viability of the algebraic mapping operators without it impeding the performance of a mapping engine. Finally, we conclude and discuss future work in Section 7.

2. Related Works

In this section, we provide an overview of existing mapping languages (Section 2.1) and existing formalizations (Section 2.2), and provide concluding discussions (Section 2.3).

2.1. Mapping Languages

Declarative mapping languages allow describing how to generate a knowledge graph by mapping existing data. These languages can be categorized based on the number of input data source types that they support: i) homogeneous mapping languages or ii) heterogeneous mapping languages.

On the one hand, *homogeneous mapping languages* only support one input data source type. For example, relational databases to RDF with languages like D2RQ [19], R2RML [1] and SML [2], or CSV data to RDF with TARQL [20].

On the other hand, *heterogeneous mapping languages* support multiple input data source types, i.e., multiple data formats (CSV, JSON, XML, etc...) and multiple data access (websocket, files, databases, etc...). Heterogeneous mapping languages can be further categorized [8]: i) *query-based mapping languages*, ii) *dedicated mapping languages*, and iii) *constraint-based mapping languages*.

Query-based mapping languages extend or adapt the SPARQL syntax to map heterogeneous data to a KG, e.g. XSPARQL [21] combines XQuery and SPARQL, SPARQL-Generate [5] extends SPARQL with generation-specific operators, and SPARQL-Anything [17] applies extended SPARQL (CONSTRUCT) queries over an input data meta-model called Façade-X [22].

Dedicated mapping languages extend existing mapping languages or use custom syntax. RML [3, 7], xR2RML [23], and D2RML [24] are examples of dedicated mapping languages which extend R2RML to support more than just relational databases. Amongst them, RML is the most matured mapping language, taken up by the W3C KG-Construct Community Group[1] for ongoing standardization [7]. D-REPR [4] is the only dedicated mapping language with its own syntax.

Currently, ShExML [6] is the only *constraint-based* mapping language based on the ShEx [25] syntax with extensive modifications and with a focus on making a user-friendly language. Although ShExML itself does not use constraints internally during the mapping of data to RDF, we follow the categorization of Van Assche et al. [8].

2.2. Formalizations

Several formalization works have been conducted to formalize the aforementioned languages. For *homogeneous mapping languages*, Stadler et al. [2] provide a unified model, focussed on relational databases as input data source type. However, there are 2 limitations to their approach. First, the formalizations are only applied to the authors' own mapping language, SML. Last, the translation of R2RML to SML – to demonstrate the completeness of their unified model – is informally described. Therefore, it is uncertain if the model is suitable for providing operational semantics for a generic mapping process. Priyatna et al. [26] provide formalizations for the translation of SPARQL to SQL based on the mapping definitions in an R2RML document, extending the works of Chebotko et al. [27] and improving the works of SparqlMap [28]. Since the formalization is not applied directly upon R2RML, it only provides partial operational semantics to R2RML.

For *heterogeneous mapping languages*, on the one hand, query-based mapping languages benefit from the usage of SPARQL semantics, which results in having partial operational semantics out-of-the-box. XSPARQL [21] provides partial operational semantics on the combination of XQuery and SPARQL, using SPARQL's semantics. Similarly, SPARQL-Anything and SPARQL-Generate extend the existing SPARQL syntax to ensure that their operational semantics inherit SPARQL's well-defined semantics [18]. For SPARQL-Anything, it formally describes the heterogeneous input data with their RDF meta-model Façade-X [22], and uses SPARQL to query the RDF meta-model [17], leading to SPARQL-Anything having provided formal operational semantics of its mapping process. SPARQL-Generate [5] also provided the operational semantics for KG generation from heterogeneous data based on its extended SPARQL syntax.

On the other hand, dedicated mapping languages such as RML requires the authors to analyze the mapping language syntax and formulate their own operational semantics. RML Fields [29] extended RML's syntax to also allow mapping of nested heterogeneous data, provided an informal operational semantics of how it works. Iglesias et al. [15], and Arenas et al. [9] provide formalizations for RML, used to optimize the mapping process by grouping the execution order using the concept of *mapping assertions*, and *mapping partitions* respectively. *Mapping assertions* are formalized using Horn clauses, whereas *mapping partitions* are formalized using set theory. This mismatch on the formalizations techniques makes it difficult to determine the similarity between the two optimization approaches. The formalizations employed in both works are successfully used to prove the optimization techniques, however, they are tailored to RML and do not to introduce general operational semantics of a mapping process.

[1] https://www.w3.org/community/kg-construct/

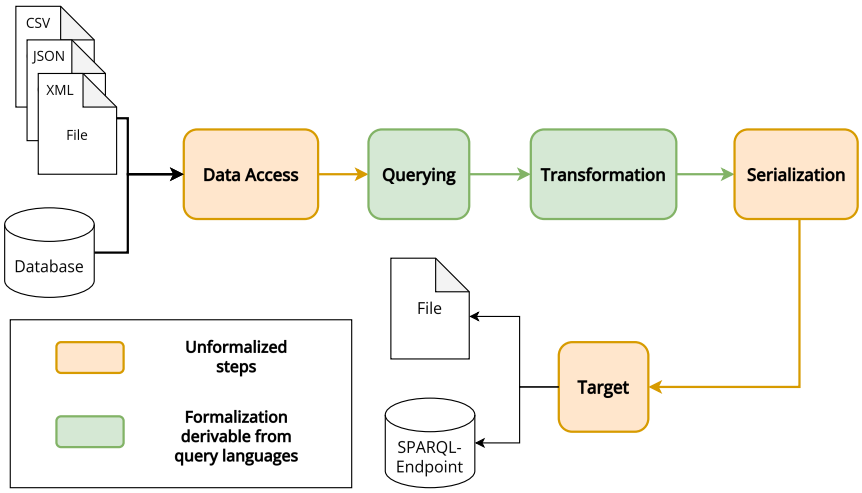

Figure 1. Breakdown of the mapping process: the *green* labelled steps can be derived from existing query algebra, and the *orange* labelled steps need to be formally defined. The exemplary Data Access step can read data from a relational database, and a CSV file while the exemplary Target step, writes the generated output to both a file and a SPARQL endpoint.

2.3. Discussion

We observe that existing formalizations are typically partial, in the case of query-based languages partially relying on SPARQL semantics, and exclusively applicable to a specific syntax: they do not provide a unified model for a generic mapping process independent of the mapping language. Currently, there is only a single work on providing a unified model for mapping languages through an ontology [14]. However, it is not formal and does not provide operational semantics. Hence, no KG generation operational semantics independent of the mapping language currently exists.

The lack of such operational semantics impedes using an optimization technique proposed in one mapping language in another mapping language. Furthermore, static analysis cannot be executed for verification of the mapping rules described using these languages. For example, mapping engines such as SDM-RDFizer [15] and Morph-KGC [9] employ their own concept of operational semantics for optimizing the mapping process. This makes it impossible to formally verify the combination of both techniques without a translation between the two operational semantics.

3. Methodology

In this section, we discuss our methodology for developing a mapping algebra and the rationale behind our design choices in selecting a specific set of algebraic mapping operators to represent our mapping algebra.

A mapping process can typically be decomposed into the followings steps: i) *data access*, ii) *data querying*, iii) *data transformations*, iv) *data serialization*, and v) *target output*. Figure 1 shows the breakdown of the mapping process.

One way to define the operational semantics of the mapping process is through the definitions of algebraic operators, similar to the query languages. It is beneficial to extend and adapt the SPARQL algebra with new algebraic operators as a foundation for defining algebraic mapping operators in a mapping algebra: i) we already have query-based mapping languages successfully leveraging SPARQL's semantics (Section 2.2), and ii) this approach allows to align dedicated and query-based mapping languages, allowing for a generic mapping plan that can support multiple mapping languages.

To apply the SPARQL algebra approach, we must define the corresponding algebraic operators for the 5 steps in the mapping process (Figure 1). Formal operational semantics of the *querying* and *transformation* operators are

well-defined in querying languages such as SPARQL [18, 30] and SQL. Querying operators are defined, in this work, as operators that do not change the *value* associated with an attribute of a data record, in contrast to the transformation operators, which derive new *values* through operations such as string concatenation. An example for a querying operator is the SPARQL's *projection* operator which we redefine Section 4.3 with minor modifications. The same is also done for other querying operators such as *join*, and *rename* in Section 4.7 and 4.5 respectively. Similarly, we adapt SPARQL's *extend* operator (Section 4.4) to provide operational semantics for the transformation step. The remaining steps – *data access*, *data serialization*, and *target output* – need to be defined formally with an extended set of algebraic operators which we define in Section 4.2, 4.8, and 4.9 respectively. As shown in Figure 1, a target operator can write the generated RDF into multiple data sinks, such as a file or a SPARQL endpoint. In the original SPARQL algebra, there exists no algebraic operators which could do a *fan-out* operation and direct the output to multiple downstream operators. Thus, we extended SPARQL algebra with the concept of *fragment* (Definition 1) and the *fragmenter operator* (Section 4.6) for handling this situation.

4. Mapping Algebra

In this section, we introduce our mapping algebra. We first describe the needed terms and then define the algebraic mapping operators: *Source*, *Projection*, *Extend*, *Rename*, *Fragmenter*, *Natural join*, *θ-join*, *Left outer-join*, *Union*, *Serialize*, and *Target*. The algebra described in this section substantially extends our previous work [31], introduces seven more operators, and improves the mapping tuple definition. Since this work is inspired by SPARQL algebra, existing definitions and terms by Perez et al. [18] will be reused where possible. We only briefly introduce the notations and concepts re-used from SPARQL algebra, readers are referred to the literature for more in-depth definitions [18]. Throughout this section, examples will be provided by applying these algebraic mapping operators sequentially on a small dataset.

4.1. Preliminaries

The following pairwise disjoint infinite sets are used in the definition of mapping algebra: V (variables), I (IRIs), B (RDF blank nodes), and L (RDF literals). A *solution mapping* μ is a mapping from variables V with associated data values of type $I \cup B \cup L$. More formally, it is defined as partial function $\mu : V \rightarrow T$ with $T = I \cup B \cup L$. A multiset of solution mapping is denoted as Ω [18]. Note that the term *mapping* in the solution mapping follows the mathematical notion of a function in the general sense: if f is a *mapping* function $f : X \rightarrow Y$, then f is a subset of $X \times Y$ consisting of all pairs $(x, f(x))$ for all $x \in X$ and that $f(x) \in Y$ [32]. This notion is different from the domain of mapping languages in KG construction community where *mapping* refers to the mapping of data in a specific format to RDF.

Two solution mappings μ_1 and μ_2 are *compatible*, if and only if, $\forall v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$; extending this, the union of μ_1 and μ_2, $\mu_1 \cup \mu_2$, is also a solution mapping. Given two multisets of solution mappings Ω_1 and Ω_2, SPARQL algebra [18] defined the *join* ($\bowtie$), the *union* ($\cup$), and the *difference* ($\setminus$) between Ω_1 and Ω_2 as follows [18]:

$$\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\} \tag{1}$$

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible. }\} \tag{2}$$

$$\Omega_1 \setminus \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 \text{ and } \mu_2 \text{ are not compatible. }\} \tag{3}$$

Now, we are ready to define the tuple type that our algebraic operators will operate upon. Unlike querying in SPARQL, mapping languages enable users to *fragment* the generated data into different data sinks (e.g. multiple files or web sockets). For example, using Logical Targets [33], RML engines can export the generated RDF output to different data sinks based on the description provided in the Logical Targets. In contrast, query languages do not allow users to specify where to export the queried data. Thus, we introduce the concept of *fragments* [31].

Table 1

Two mapping tuples describing information related to John. The tuples are fragmented according to *personal* information about John and information about John's *friends*

	Multiset of Solution Mappings			
Fragment	Solution Mapping	?name	?age	?email
$f_{personal}$	μ_1	John Doe	23	john.doe@example.com
$f_{friends}$	μ_2	Susan Sue	25	susan.sue@example.com
$f_{friends}$	μ_3	Alice Joe	26	alice.joe@example.com

Definition 1. Let Ω be a multiset of solution mappings. A **fragment**, $f \in F$, is a grouping of a submultiset [34] of Ω. The set of fragments, F, is infinite and pairwise disjoint with the other sets, V, I, B, and L, defined in Section 4.1.

Using the definition of *fragments*, the core data model of the mapping algebra, the *mapping tuple* is defined as follows.

Definition 2. Let Ω be a multiset of solution mappings, and $P(\Omega)$ the powerset of the multiset Ω [34]. A **mapping tuple**, t, is a partial function which maps *fragments* to multisets of solutions mappings $t : F \rightarrow P(\Omega)$. A multiset of mapping tuples is denoted as Γ. Note, from now on, we shall use the notation ω, as an element of the powerset $P(\Omega)$ with $\omega \in P(\Omega)$, to make the definitions of the algebraic operators more accessible to read. In this case, ω itself is also a multiset of solution mappings.

Utilizing fragments in the mapping tuple enables mapping processes to broadcast solution mappings across multiple downstream operators. Furthermore, it enables the partitioning of the solution mappings during construction based on either user defined conditions or some abstract concept such as personal or friend's information as shown in Table 1.

4.2. Source

In an algebraic mapping plan, the *source* operators are the leaf nodes of the mapping plan: they generate the required mapping tuples for further processing by the downstream algebraic operators.

A way to extract data records from heterogeneous data formats needs to be defined to generate the mapping tuples. Especially, when the input data has a nested data structure. ShExML [6] enables data records extraction from nested data structures, using *iterators* and *fields*. Furthermore, it also enables referencing of data records on different hierarchical level through the use of the *pushed* and the *popped* fields[2]. RML Fields [29] expanded upon the concept of iterators and fields used in ShExML to also allow data records extraction from nested heterogeneous data formats. For example, RML Fields can extract a JSON data record in a CSV table cell using a nested *reference formulation*, unlike ShExML where only nested data structures of the same data format is supported. This work has been recently continued as RML Logical Views[3].

To support the extraction of data records from a nested data structure, we define the *iterators* and *fields*, as part of the *source* operator, as follows. The definition is similar to the work of RML Fields.

Definition 3. An **iterator**, I, consists of one or more **fields**, $\phi^{r,v_{alias}}$, an iterator *path*, and a *reference formulation*. An iterator extracts a list of records from the data source according to the given iterator *path* [29], while the *reference formulation* determines the data format [3] of the data source. This is the entry point to further extract nested data structures with *fields*.

Definition 4. Given an iterator I, a reference path r, an *alias name* v_{alias}, an optional *reference formulation* and zero or more fields $\emptyset \subseteq \Phi$. **Fields** applied on an iterator I and zero or more fields Φ as $\phi^{r,v_{alias}}(I, \Phi)$, uses the given *reference path* r to generate one or more records from each record generated by the iterator I. The *alias name* v_{alias}

[2] ShExML pushed and popped fields: https://shexml.herminiogarcia.com/spec/#pushed-and-popped-fields
[3] RML Logical Views: https://github.com/kg-construct/rml-lv.

of the field is used as a variable to generate a solution mapping μ such that $\mu(v_{alias})$ is the data record extracted by $\phi^{r,v_{alias}}(I,\Phi)$. The *reference formulation* is optionally used to determine the data format of the part of the data record the associated field is extracting. This enables extraction of data records with mixed data formats in nested data structures as defined in RML Fields. The set of fields Φ, are the subfields that further extract nested data structures at a deeper hierarchical level.

Definition 5. Given a data access configuration C_{access}, and an iterator I which consists of one or more fields $\emptyset \nsubseteq \Phi$. The **source** operator generates a multiset of mapping tuples, $t \in \Gamma$. Since the generated mapping tuples do not belong to a fragment yet, we define a default fragment, f_0, as the fragment to which **all** the mapping tuples initially belong to. Data access configuration C_{access} consists of metadata information about utilizing the data source (e.g. connection ports and credentials for Apache Kafka[4]). Iterators enable querying of the data source to extract data records and generate our solution mappings μ. The field names are used as the variables, and the associated extracted data value is generated as a *Literal*. The extracted data value can have a datatype in the *Literal* if the datatype can be inferred from the data source. This mapping between source data values and corresponding RDF datatypes are (sometimes implicitly) defined per mapping language[5]. For example, in RML, when extracting a data value from a JSON file typed as a JSON boolean, the extracted data value will have the corresponding xsd:boolean datatype. This is to align the data records with the definition of solution mappings $\mu : V \rightarrow T$.

$$\begin{aligned} f_0 &= \text{a default fragment} \\ \mu &= \text{flattened data record iterated according to } I \text{ and fields } \Phi \\ \omega &= \text{a multiset containing } \mu \\ \text{Source}(C, I) &= \{t \mid t = f_0 \rightarrow \omega\} \end{aligned} \tag{4}$$

To clarify the workings of the source operator, iterator and fields, Example 1 shows how mapping tuples are generated from a simple JSON file with nested data using a source operator with its iterator and fields.

Listing 1: Example input data in JSON format

```
{
    "people": [
        {
            "name" :               "John Doe",
            "age"  :                     23,
            "email": "john.doe@example.com",
            "pet"  :
            {
                "type": "dog",
                "name": "Bax"
            }
        },
        {
            "name" :               "Susan Sue",
            "age"  :                     23,
            "email": "susan.sue@example.com",
        }
    ]
}
```

Example 1. As an example, provided with an input data source such as the JSON file in Listing 1, the source operator could be configured with the following C and I:

- C: path to the JSON file.
- I: with reference formulation "JSONPath" and iterator path $.people[*]

[4] Apache Kafka: https://kafka.apache.org/

[5] For an explicit example, see https://kg-construct.github.io/rml-io-registry/json-path/#natural-rdf-mapping

Table 2
Representation of the mapping tuples generated by the source operator using data from Listing 1

	Multiset of Solution Mappings					
Fragment	Solution Mapping	?name	?age	?email	?pet.type	?pet.name
$f_{default}$	μ_1	John Doe	23	john.doe@example.com	dog	Bax
$f_{default}$	μ_2	Susan Sue	25	susan.sue@example.com		

The iterator I also contains the following fields Φ applied as:

- $\phi_1^{r,v_{alias}}(I,\emptyset)$: with $r =$ \$.name and $v_{alias} =$ "?name"
- $\phi_2^{r,v_{alias}}(I,\emptyset)$: with $r =$ \$.age and $v_{alias} =$ "?age"
- $\phi_3^{r,v_{alias}}(I,\emptyset)$: with $r =$ \$.email and $v_{alias} =$ "?email"
- $\phi_4^{r,v_{alias}}(I,\{\phi_5^{r,v_{alias}},\phi_6^{r,v_{alias}}\})$: with $r =$ \$.pet and $v_{alias} =$ "?pet"
- $\phi_5^{r,v_{alias}}(I,\emptyset)$: with $r =$ \$.name and $v_{alias} =$ "name"
- $\phi_6^{r,v_{alias}}(I,\emptyset)$: with $r =$ \$.type and $v_{alias} =$ "type"

The iterators are using the JSONPath reference formulation[6] [35]. The source operator generates a mapping tuple as shown in Table 2 with a default fragment $f_{default}$. The fields are executed relative to the iterator, I, and assign the extracted value to the variable described by the name of the field.

Execution of $\phi_4^{r,v_{alias}}(I,\{\phi_5^{r,v_{alias}},\phi_6^{r,v_{alias}}\})$ provides the context for the execution of both $\phi_5^{r,v_{alias}}(I,\emptyset)$ and $\phi_6^{r,v_{alias}}(I,\emptyset)$ such that two data records are generated for the variables "?pet.name" and "?pet.type" respectively.

After the source operator generates the mapping tuples from heterogeneous data sources, these mapping tuples are further processed and transformed by the intermediate algebraic mapping operators. The intermediate algebraic mapping operators are defined as follows: *Projection*, *Rename*, *Extend*, *Fragmenter*, and the various *Join* operators.

4.3. Projection

In SPARQL algebra, a *projection* restricts solution mappings to a set of variables. This is useful in reducing the amount of data that needs to be further processed by the downstream operators. The corresponding *projection* operator in the mapping algebra is defined as follows.

Definition 6. Given a set of variables $P \subseteq V$, the **projection** operator restricts the variables in the solution mappings μ, associated with the mapping tuple t, according to P.

$$\begin{aligned}
\text{Project}(\mu, P) &= \mu \text{ restricted to variables in } P \\
\text{Project}(\omega, P) &= \{\text{Project}(\mu, P) \mid \forall \mu \in \omega\} \\
\text{Project}(t, P) &= \{(f, \text{Project}(\omega, P)) \mid \forall (f, \omega) \in t\} \\
\text{Project}(\Gamma, P) &= \{\text{Project}(t, P) \mid t \in \Gamma\}
\end{aligned} \tag{5}$$

Example 2. A projection operator, configured with a set of variables {"?name", "?pet.name", "?pet.type"} $\in P$, applied on the generated mapping tuple in Table 2 generates a mapping tuple in Table 3.

4.4. Extend

A core operation of the mapping process is the derivation of new values using existing values in the data record. For example, given a data record containing the weight and height of a person, we can calculate the body mass index (BMI) of the person using their weight and height. The following definition of the *extend* operator enables the mapping algebra to derive new values from existing values.

[6]

Table 3
Projection operator from Example 2 applied on the mapping tuple shown in Table 2

	Multiset of Solution Mappings			
Fragment	Solution Mapping	**?name**	**?pet.type**	**?pet.name**
$f_{default}$	μ_1	John Doe	dog	Bax
$f_{default}$	μ_2	Susan Sue		

Table 4
Extended mapping tuple as described in Example 3

	Multiset of Solution Mappings				
Fragment	Solution Mapping	?name	?pet.type	?pet.name	**?name_iri**
$f_{default}$	μ_1	John Doe	dog	Bax	<http://example.com/John%20Doe>
$f_{default}$	μ_2	Susan Sue			<http://example.com/Susan%20Sue>

Definition 7. Given a set of pairs $(v_{new}, expr) \in E$, with $v_{new} \notin dom(\mu), v_{new} \in V$ and $expr : \Omega \rightarrow T$ an extend expression. The **extend** operator derives a new value, *value* by evaluating the extend expression $expr$ on the solution mapping ω such that $expr(\omega) = value$. It extends the solution mapping with the generated value, which is coupled to the new variable v_{new} such that $\omega(v_{new}) = value$. If evaluating $expr$ causes an error and $v_{new} \notin dom(\mu)$, the extend operator behaves like an identity operator. It is undefined if the variable restriction is violated, which means $v_{new} \in dom(\mu)$. Formally, it is defined as follows.

$$\begin{aligned} \text{Extend}(\mu, E) &= \mu \cup \{(v_{new}, value) \mid (v_{new}, expr) \in E, v_{new} \notin \text{dom}(\mu) \text{ and value} = \text{expr}(\mu)\} \\ \text{Extend}(\omega, E) &= \{\text{Extend}(\mu, E) \mid \forall \mu \in \omega\} \\ \text{Extend}(t, E) &= \{(f, \text{Extend}(\omega, E)) \mid \forall (f, \omega) \in t\} \\ \text{Extend}(\Gamma, E) &= \{\text{Extend}(t, E) \mid t \in \Gamma\} \end{aligned} \tag{6}$$

Different mapping languages can introduce different extend expressions.

Example 3. Provided $\{(?name\_iri, encodeIri^{?name})\} \in E$ with $encodeIri^{?name}$ a (custom RML) expression that returns an IRI using the $?name$ attribute from a given solution mapping μ by IRI encoding the data value. The extend operator applied to the mapping tuple shown in Table 3 generates the mapping tuple shown in Table 4.

4.5. Rename

In order to avoid variable collision when processing mapping tuples, an algebraic operator must be able to rename the variables inside the solution mappings associated with the mapping tuples. The *rename* operator, which introduces aliasing of the existing variables in the solution mappings, is defined as follows.

Definition 8. Given a set of pairs of variables $\{(v_{a1}, v_{b1})..(v_{an}, v_{bn})\} \in R$. The **rename** operator, applied on a multiset of mapping tuples Γ, renames the variables of the solution mappings associated with the mapping tuples as follows: if $\mu \in range(t)$, for each $(v_a, v_b) \in R$ rename $v_a \rightarrow v_b$ if $v_a \in dom(\mu)$. If the rename operator is configured with an alias string s_{alias} instead of R, the rename operator will concatenate s_{alias} as the suffix for all $v \in dom(\mu)$ as $s_{alias} \| v$.

Table 5
Output of the rename operator as described in Example 4

	Multiset of Solution Mappings				
Fragment	Solution Mapping	**?fullname**	?pet.type	?pet.name	?name_iri
$f_{default}$	μ_1	John Doe	dog	Bax	<http://example.com/John%20Doe>
$f_{default}$	μ_2	Susan Sue			<http://example.com/Susan%20Sue>

$$
\begin{aligned}
\text{Rename}(\mu, R) &= \{(v_b, d) \mid \forall (v_a, v_b) \in R, \forall (v, d) \in \mu, v = v_a\} \cup \\
&\quad \{(v, d) \mid \forall (v_a, v_b) \in R, \forall (v, d) \in \mu, v \neq v_a\} \\
\text{Rename}(\omega, R) &= \{\text{Rename}(\mu, R) \mid \forall \mu \in \omega\} \\
\text{Rename}(t, R) &= \{(f, \text{Rename}(\omega, R)) \mid \forall (f, \omega) \in t\} \\
\text{Rename}(\Gamma, R) &= \{\text{Rename}(t, R) \mid t \in \Gamma\}
\end{aligned} \tag{7}
$$

$$
\begin{aligned}
\text{Rename}(\mu, s_{alias}) &= \{(v_{aliased}, d) \mid \forall (v, d) \in \mu, v_{aliased} = s_{alias} \| v\} \\
\text{Rename}(\omega, s_{alias}) &= \{\text{Rename}(\mu, s_{alias}) \mid \forall \mu \in \omega\} \\
\text{Rename}(t, s_{alias}) &= \{(f, \text{Rename}(\omega, s_{alias})) \mid \forall (f, \omega) \in t\} \\
\text{Rename}(\Gamma, s_{alias}) &= \{\text{Rename}(t, s_{alias}) \mid t \in \Gamma\}
\end{aligned} \tag{8}
$$

Readers would also realize that the *rename* operator can also be derived by first extending the solution mapping with a new variable (Extension), copying the value associated with the old variable, and finally projecting away the old variable (Projection). We defined the *rename* operator to describe the execution of the rename operation in one operator instead of two operators. This reduces the complexity and redundancy of the generated mapping plan using the operators. The *extend* and *project* operator chaining to represent the rename operation is formally defined as follows.

$$
\begin{aligned}
R &= \{(v_{a1}, v_{b1}) \ldots (v_{an}, v_{bn} \mid n \in \mathbb{N}\} \\
P_{renamed} &= \{v \mid v \in dom(\mu)\} \cup \{v_b \mid \forall (v_a, v_b) \in R\} / \{v_a \mid \forall (v_a, v_b) \in R\} \\
copyData(v) &= \text{copies the value of } \mu(v) \text{ when evaluated by the extend operator on a solution mapping } \mu \\
E_{rename} &= \{(v_b, copyData(v_a)) \mid \forall (v_a, v_b) \in R\} \\
\text{Rename}(\Gamma, R) &= \text{Project}(\text{Extend}(\Gamma, E_{rename}), P_{renamed})
\end{aligned} \tag{9}
$$

Example 4. Provided with a set of variable pairs $\{(?name, ?fullname)\} \in R$. Applying the rename operator on the mapping tuples in Table 4, generates the mapping tuples in Table 5. The old variable, $?name$, in the solution mappings is renamed to $?fullname$.

4.6. Fragmenter

The aforementioned operators do not manipulate the *fragments* part of the mapping tuples, but only process the associated solution mappings. To manipulate the fragments of the mapping tuples, the *fragment* operator is defined as follows.

Table 6
Fragmentation of mapping tuple as described in Example 3

	Multiset of Solution Mappings				
Fragment	Solution Mapping	?fullname	?pet.type	?pet.name	?name_iri
$f_{contacts}$	μ_1	John Doe	dog	Bax	<http://example.com/John%20Doe>
$f_{contacts}$	μ_2	Susan Sue			<http://example.com/Susan%20Sue>

Definition 9. The **fragmenter** operator fragments the mapping tuple into a new fragment f_{new}. Given a partial transformation function, $\delta : F \rightarrow F$. A fragmenter operator applies δ on the mapping tuple $t = f \rightarrow \mu$, and map it into a new mapping tuple $t_{mapped} = f_{new} \rightarrow \mu$ if $dom(\delta) \subseteq dom(t)$ and $f_{new} \in range(\delta)$. When $dom(\delta) \nsubseteq dom(t)$, fragmenter operator acts like an identity function. More formally, it is defined as follows.

$$
\begin{aligned}
\delta &= \{(f_{old}, f_{new}) \mid f_{old}, f_{new} \in F\} \\
\delta(t) &= \{(f, \omega) \mid (f, \omega) \in t, f \neq f_{old}\} \cup \{(f_{new}, \omega) \mid (f_{old}, \omega) \in t, (f_{old}, f_{new}) \in \delta\} \\
\text{Fragment}(t, \delta) &= \begin{cases} \delta(t) & \text{if } dom(\delta) \subseteq dom(t) \\ t & \text{otherwise} \end{cases} \\
\text{Fragment}(\Gamma, \delta) &= \{\text{Fragment}(t, \delta) \mid t \in \Gamma\}
\end{aligned}
\tag{10}
$$

Example 5. Continuing with the output of the *extend* operator as shown in Table 4. A fragmenter operator with $\delta : \{(f_{default}, f_{contacts})\}$ applied on the mapping tuples generates new mapping tuples shown in Table 6, where the old fragment, $f_{default}$, is mapped to the new fragment, $f_{contacts}$.

4.7. Binary operators

Previously defined operators are unary: they only work on a single multiset of mapping tuples Γ. To combine two multisets of mapping tuples Γ_1, and Γ_2, binary algebraic operators need to be defined, along with the definition of *compatibility* between the mapping tuples. Thus, to support the binary operations for combining mapping tuples, the mapping algebra defines the *compatibility of mapping tuples*, the *natural join*, the *θ-join*, and the *left-join* between Γ_1 and Γ_2 as follows.

Definition 10. Two mapping tuples, $t_1 \in \Gamma_1$ and $t_2 \in \Gamma_2$, are **compatible**, if and only if, $\forall f \in dom(t_1) \cap dom(t_2)$, for the associated multisets of solution mappings $t_1(f) = \Omega_1$ and $t_2(f) = \Omega_2$, $\exists \mu_1 \in \Omega_1, \exists \mu_2 \in \Omega_2$ where μ_1 and μ_2 are *compatible* so that $\Omega_1 \bowtie \Omega_2 \neq \emptyset$. See Section 4.1 for solution mappings compatibility. If $dom(t_1)$ and $dom(t_2)$ are disjoint, t_1 and t_2 are compatible.

4.7.1. Natural join

Definition 11. Natural join is a binary operator that combines two multisets of mapping tuples Γ_1 and Γ_2 if they are *compatible* (Definition 10). It produces mapping tuples, $t_1 \bowtie t_2$, which is a combination of two mapping tuples, $t_1 \in \Gamma_1$ and $t_2 \in \Gamma_2$ that have common fragments, $\forall f \in dom(t_1) \cap dom(t_2)$, for which the associated multisets of solution mappings $\Omega_1 = t_1(f)$ and $\Omega_2 = t_2(f)$ are joined as $\Omega_1 \bowtie \Omega_2$ according to Equation 2. It is formally defined as follows.

$$
\begin{aligned}
\text{NatJoin}(t_1, t_2) &= \{(f, \Omega_1 \bowtie \Omega_2)) \mid \forall f \in dom(t_1) \cap dom(t_2), t_1(f) = \Omega_1, t_2(f) = \Omega_2\} \\
\text{NatJoin}(\Gamma_1, \Gamma_2) &= \{\text{NatJoin}(t_1, t_2) \mid t_1 \in \Gamma_1, t_2 \in \Gamma_2, t_1 \text{ and } t_2 \text{ are compatible}\}
\end{aligned}
\tag{11}
$$

Natural join operator joins mapping tuples if they have common *fragments*, and they also have equal values for all the common *variables* of the associated solution mapping of the fragment. It does not allow users to join mapping tuples based on the different variables of $dom(\mu_1)$ and $dom(\mu_2)$. Furthermore, natural join only checks for the *equality* condition on the common variables and fragments: it does not use other predicate functions such as $\leqslant$ or $\geqslant$.

4.7.2. θ-join

θ-join enables the use of a predicate function, θ, on the specified variables to join two multisets of mapping tuples. Thus, it is a more general form of the natural join operator. In order to execute θ-join, the following conditions must be satisfied: $\forall v_a \in \mu_a \ \wedge \ \forall v_b \in \mu_b.\ v_a \neq v_b$. Otherwise, μ_a and μ_b can be incompatible and cannot be joined as $\mu_a \cup \mu_b$ (see Section 4.1 for compatibility definition). Thus, to make sure that μ_a and μ_b are compatible, the mapping plan planner should apply the rename operator before the θ-join operator to ensure that none of variables in μ_a and μ_b are equal to each other. It is defined as follows.

Definition 12. Given a binary predicate function, ${}_{v_1}\theta_{v_2} : \Omega \times \Omega \rightarrow boolean$, with variables v_1, and v_2, where $v_1 \in dom(\mu_1), v_2 \in dom(\mu_2)$, and two multisets of mapping tuples Γ_1 and Γ_2. Provided that $\forall v_a \in dom(\mu_1), \forall v_b \in dom(\mu_2), v_a \neq v_b$. θ-join then combines two mapping tuples $t_1 \in \Gamma_1$, and $t_2 \in \Gamma_2$, if for the same fragment $f \in dom(t_1) \cap dom(t_2)$, the following condition is satisified: $\mu_1 \in \Omega_1, \mu_2 \in \Omega_2, {}_{v_1}\theta_{v_2}(\mu_1, \mu_2) = true$ with $\Omega_1 = t_1(f)$ and $\Omega_2 = t_2(f)$.

$$
\begin{aligned}
{}_{v_1}\theta_{v_2}(\mu_1, \mu_2) &= \begin{cases} true, & \text{if } \mu_1(v_1) = \mu_2(v_2) \\ false, & \text{otherwise} \end{cases} \\
\text{ThetaJoin}(\Omega_1, \Omega_2, {}_{v_1}\theta_{v_2}) &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2, {}_{v_1}\theta_{v_2}(\mu_1, \mu_2) \text{ evaluates to true } \} \\
\text{ThetaJoin}(t_1, t_2, {}_{v_1}\theta_{v_2}) &= \{(f, \text{ThetaJoin}(\Omega_1, \Omega_2, {}_{v_1}\theta_{v_2})) \mid \forall f \in dom(t_1) \cap dom(t_2), \\
&\qquad\qquad t_1(f) = \Omega_1, t_2(f) = \Omega_2\} \\
\text{ThetaJoin}(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2}) &= \{\text{ThetaJoin}(t_1, t_2, {}_{v_1}\theta_{v_2}) \mid t_1 \in \Gamma_1, t_2 \in \Gamma_2\}
\end{aligned}
\tag{12}
$$

Natural and θ-join filters out solution mappings which do not satisfy the predicate function. The filtering of solution mappings results in the loss of information which might require extra processing steps to retain them. Thus, a new binary algebraic operator, which retains the solution mappings that do not satisfy the given predicate function, needs to be defined. SPARQL and relational algebra have definitions for such a group of binary operators called *outer-joins*.

4.7.3. Left outer-join

Left outer-join operator is a binary operator that retains the solution mappings from the *left* multisets of mapping tuples even if the given solution mappings do not satisfy the predicate function. Other outer-join operators, such as *right outer-join*, and *full outer-join* operators, can be derived from the left outer-join operator.

Definition 13. Given two multisets of mapping tuples, Γ_1 and Γ_2, with $t_1 \in \Gamma_1, t_2 \in \Gamma_2, \Omega_1 \in t_1, \Omega_2 \in t_2, \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$, and a predicate function, ${}_{v_1}\theta_{v_2}$. Similar to *θ-join*, **left outer-join** requires the application of a rename operator beforehand to ensure compatibility between the solution mappings. If Γ_1 and Γ_2 are *incompatible*, **left outer-join** keeps the mapping tuples from Γ_1 but drops everything from Γ_2. If Γ_1 and Γ_2 are *compatible*, **Left outer-join** combines two multisets of mapping tuples, Γ_1 and Γ_2 based on the *boolean condition* after evaluating ${}_{v_1}\theta_{v_2}(\mu_1, \mu_2)$ as follows. If it is true, it behaves the same as the *θ-join* operator producing $\mu_1 \cup \mu_2$ for the associated mapping tuple t. Otherwise, only μ_1 is added to the mapping tuple t while μ_2 is dropped. Thus, **left outer-join** operator can be broken down into a taking a union of two steps: the union of the θ-join operator and the *difference* operator. The *difference* operator is used internally to define the left outer-join operator, similar to the definition of left-join in SPARQL algebra [18]. More formally, it is defined as follows.

Table 7
Mapping tuples generated from another data source about pets

	Multiset of Solution Mappings			
Fragment	Solution Mapping	?type	?name	?age
$f_{contacts}$	μ_{a1}	dog	Bax	10
$f_{contacts}$	μ_{a2}	cat	Coco	3
$f_{contacts}$	μ_{a3}	dog	Max	5

Table 8
Output of the natural join operator as described in Example 6. Only the common variables *?pet.name* and *?pet.type* are checked.

	Multiset of Solution Mappings					
Fragment	Solution Mapping	?fullname	**?pet.type**	**?pet.name**	?name_iri	?pet.age
$f_{contacts}$	$\mu_1 \cup \mu_{a1}$	John Doe	dog	Bax	<http://example.com/John%20Doe>	10

$$
\begin{aligned}
\text{Difference}_\theta(\Omega_1, \Omega_2, {}_{v_1}\theta_{v_2}) &= (\Omega_1 \setminus \Omega_2) \cup \{\mu_1 \mid \mu_1 \in \Omega_1, \exists \mu_2 \in \Omega_2, {}_{v_1}\theta_{v_2}(\mu_1, \mu_2) \text{ evaluates to } false\ \} \\
\text{Difference}_\theta(t_1, t_2, {}_{v_1}\theta_{v_2}) &= \{(f, \text{Difference}_\theta(\Omega_1, \Omega_2, {}_{v_1}\theta_{v_2})) \mid \forall f \in dom(t_1) \cap dom(t_2), \\
&\qquad \Omega_1 = t_1(f), \Omega_2 = t_2(f)\ \} \\
&\quad \cup \{(f, \omega) \mid \forall f \in dom(t_1) \setminus dom(t_2), \omega = t_1(f)\} \\
\text{Difference}_\theta(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2}) &= \{\text{Difference}_\theta(t_1, t_2, {}_{v_1}\theta_{v_2}) \mid t_1 \in \Gamma_1, t_2 \in \Gamma_2, t_1 \text{ and } t_2 \text{ are compatible. }\} \\
\text{Difference}(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2}) &= \{t_1 \mid t_1 \in \Gamma_1, \forall t_2 \in \Gamma_2,\ t_1 \text{ and } t_2 \text{ are not compatible}\} \cup \\
&\qquad \text{Difference}_\theta(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2}) \\
\text{LeftJoin}(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2}) &= \text{ThetaJoin}(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2}) \cup \text{Difference}(\Gamma_1, \Gamma_2, {}_{v_1}\theta_{v_2})
\end{aligned}
\tag{13}
$$

We provide the following three examples, where the three aforementioned join operators are applied on the mapping tuples shown in Table 6 and Table 7 as Γ_1 and Γ_2 respectively.

Example 6. Since natural join assumes solution mappings to have common variables, this example adjusts the solution mappings in Table 7 by renaming the variables of the solution mappings with the prefix "?*pet*.". The natural join operator joins the mapping tuples from Table 6 and the adjusted Table 7 (renamed with the suffix "?*pet*."), and it produces the output as shown in Table 8. The natural join merges solution mappings based on the value of the common variables. In the example, the common variables between the two different multisets of solution mappings are *?pet.name* and *?pet.type*. Since only μ_1 and μ_{a1} have the same values for the variables *?pet.name* and *?pet.type*, the output only contains $\mu_1 \cup \mu_{a1}$ and drops the other solution mappings.

Example 7. To satisfy the precondition that the variables in the solution mappings should not collide, we rename the variables in the solution mappings from Table 7 with a prefix string "*animal_*". In practice, the renaming is done by using a rename operator right before the θ-join operator. Provided with the predicate function, $?pet.type\theta?animal\_type$, for equality check on the variable *?pet.type* and the variable ?*animal_type*. θ-join operator, applied on Γ_1 and Γ_2, produces the output in Table 9. Unlike the natural join operator, the θ-join operator joins the mapping solutions only if they satisfy the conditions of the provided predicate function: in this case, an equality check on the variables *?pet.type* and ?*animal_type*.

Example 8. Provided with the predicate function, $?pet.type\theta?animal\_type$, for equality check on the variable *?pet.type* and the variable ?*animal_type*. Left outer-join operator, applied on Γ_1 and Γ_2, produces the output in Table 10.

Table 9

Output of θ-join operator as described in Example 7. Only the variables *?pet.type* and *?animal_type* are checked for equality.

	Multiset of Solution Mappings						
Fragment	Solution Mapping	?fullname	**?pet.type**	...	?animal_age	**?animal_type**	?animal_name
$f_{contacts}$	$\mu_1 \cup \mu_{a1}$	John Doe	dog	...	10	dog	Bax
$f_{contacts}$	$\mu_1 \cup \mu_{a3}$	John Doe	dog	...	5	dog	Max

Table 10

Output of left-join operator as described in Example 8. Only the variables *?pet.type* and *?animal_type* are checked for equality and all solution mappings from Γ_1 are retained.

	Multiset of Solution Mappings						
Fragment	Solution Mapping	?fullname	**?pet.type**	...	?animal_age	**?animal_type**	?animal_name
$f_{contacts}$	$\mu_1 \cup \mu_{a1}$	John Doe	dog	...	10	dog	Bax
$f_{contacts}$	$\mu_1 \cup \mu_{a3}$	John Doe	dog	...	5	dog	Max
$f_{contacts}$	μ_2	Susan Sue		...			

In this example, the left outer-join retains the solution mapping μ_2, even though it does not satisfy the predicate function. For solution mapping μ_1, it produces the same result as Table 9.

4.7.4. Union

The aforementioned join operators have a limitation where they cannot merge mapping tuples without comparing the actual data values in the solution mappings. In order to collect mapping tuples from multiple operators, without data value comparisons, we need to define a **union** operator. The algebraic mapping **union** operator is based on SPARQL's *union* operation. It is defined as follows.

Definition 14. Given two multisets of mapping tuples, Γ_1 and Γ_2. *Union* operator produces a new multiset containing mapping tuples from either Γ_1 or Γ_2. More formally, it is defined as follows.

$$\begin{aligned} Union(t_1, t_2) = &\{(f, \Omega_1 \cup \Omega_2) \mid \forall f \in dom(t_1) \cap dom(t_2), \Omega_1 = t_1(f), \Omega_2 = t_2(f)\} \\ &\cup \{(f_1, \omega) \in t_1 \mid f_1 \notin dom(t_2)\} \\ &\cup \{(f_2, \omega) \in t_2 \mid f_2 \notin dom(t_1)\} \\ Union(\Gamma_1, \Gamma_2) = &\{Union(t_1, t_2) \mid t_1 \in \Gamma_1, t_1 \in \Gamma_2\} \end{aligned} \tag{14}$$

4.8. Serialize

The aforementioned operators process and transform the mapping tuples generated from heterogeneous data sources, they do not define how to process the mapping tuples to the target data format. **Serialize** operator enables the transformation of mapping tuples to the target data format. In order to keep the operator algebraic, the output of the serializer operator is a special mapping tuple containing a solution mapping with only a single variable *?serialized_output* containing the serialized data. The serialization of the mapping tuple, according to a serialization format, is achieved through the evaluation of a specific extend expression defined as *serializer expression*. One would also notice that due to the *multiset definitions* of the mapping tuples and solution mappings, duplicate outputs can be generated if the cardinality of a solution mapping is more than one. However, this can easily be rectified with the introduction of a deduplication operator as future work.

4.8.1. Serializer expression Ψ_C

A serializer expression, $\Psi_C : \Omega \rightarrow T$, is an extend expression (Definition 7) that generates an RDF Literal containing the serialized data from a mapping tuple according to the serialization configuration C. In this work, we

Table 11
Output of the serialize operator as described in Example 9.

	Multiset of Solution Mappings	
Fragment	Solution Mapping	**?serialized_output**
$f_{contacts}$	μ_1	<http://example.com/John%20Doe> <http://example.com/name> "John Doe"; <http://example.com/petName> "Max".

focus on the generation of knowledge graphs. Thus, the serialization configuration, C, for the serializer operator is the *quad patterns* (QPs) as defined in SPARQL[6]. Since the blank nodes labels may be created using the *extend* operator and bound to a variable, we do not require the functionality to handle blank nodes separately in the serializer operator. Thus, when the serializer operator evaluates Ψ_C on a mapping tuple t, it binds for each variable, $v \in V$, in the *quad pattern* of C, with (i.e, similar to the CONSTRUCT query from SPARQL [7]).

Definition 15. Provided with the serializer expression $\Psi_C : \Omega \rightarrow T, C$ the serialization configuration, and a multiset of mapping tuples Γ. The **serialize** operator is defined using the *extend* and *projection* operator. It first applies the extend operator configured with $(v_{serialized}, \Psi_C) \in E$, Extend$(\Gamma, E)$. Afterwards, it projects the extended mapping tuples using the projection operator, Project(Γ, P) configured with $P = \{v_{serialized}\}$. More formally, it is defined as follows.

$$
\begin{aligned}
C &= \text{QPs representing the triples or quads output} \\
\Psi_C &= \text{replace the variables in the QPs based on the input solution mapping } \mu \\
E &= \{(v_{serialized}, \Psi_C)\} \\
P &= \{v_{serialized}\} \\
\text{Serialize}(\Gamma, \Psi_C) &= \text{Project}(\text{Extend}(\Gamma, E), P)
\end{aligned}
\tag{15}
$$

Note that, similar to the rename operator, the serialize operator is defined as a fixed chaining of a Projection operator (Section 4.3) after the Extend operator (Section 4.4). We defined the serialize operator to describe the execution of the serialization operation in one operator instead of two operators. This reduces the complexity and redundancy of the generated mapping plan.

Listing 2: Example QP configuration for the serialize operator

```
?name_iri <http://example.com/name> ?fullname ;
          <http://example.com/petName> ?animal_name .
```

Example 9. Provided with a QP configuration C as shown in Listing 2. *Serialize* operator evaluates the QP on each solution mappings from the mapping tuple and binds the variables $\{?name\_iri, ?fullname, ?animal\_name\}$ with the associated values. In this example, the variable $v_{serialized}$ is $?serialized\_output$. The output of the serialize operator, applied on the input mapping tuples in Table 8, is shown in Table 11.

4.9. Target

Finally, depending on the configuration of the data sink, the serialized data is written to heterogeneous data sinks such as files, websockets or Apache Kafka topics[8]. In mapping algebra, the *fragments* of the mapping tuple

[6] SPARQL Quad Pattern: https://www.w3.org/TR/sparql11-query/#rQuadPattern
[7] SPARQL Construct: https://www.w3.org/TR/sparql11-query/#construct
[8]

determine where the associated solution mappings will be written to. **Target** operator writes the data value of the specified variable, $v_{serialized}$ from the solution mapping to a target data sink associated with a particular mapping tuple with the target fragment f_{target}. If the target sink does not exist or an error occurs during the process of writing the data to the sink, the default error handling procedure of the *target* operator is to stop the whole mapping process and shows the cause of the error. An optional configuration can be provided to the target operator to change the default error handling behaviour, such as silencing the errors to continue the mapping process as much as possible.

Definition 16. Provided with a target fragment f_{target}, a target variable v_{target}, and a configuration of data sink T. **Target** operator process the mapping tuples, $t \in \Gamma$, by writing all the values $d = \mu(v_{target})$ to the data sink T, $\forall (f, \omega) \in t$ where $f = f_{target}$.

$$\begin{aligned}
\text{Target}(v_{target}, \omega, T) &= \{\text{write } d \text{ to data sink T} \mid \forall \mu \in \omega, (v, d) \in \mu, v = v_{target}\} \\
\text{Target}(f_{target}, v_{target}, t, T) &= \{\text{Target}(v_{target}, \omega, T) \mid (f, \omega) \in t, f = f_{target}\} \\
\text{Target}(f_{target}, v_{target}, \Gamma, T) &= \{\text{Target}(f_{target}, v_{target}, t, T) \mid \forall t \in \Gamma\}
\end{aligned} \tag{16}$$

Example 10. Given the input mapping tuples shown in Table 11 as Γ, and a configuration T specifying a file path */target/output.nt*. Applying **Target**$(f_{contacts}, ?serialized\_output, \Gamma, T)$ will write the serialized triples in $\mu_1(?serialized\_output)$ to the file */target/output.nt* specified by target configuration T.

5. Implementation

As a reference implementation utilizing the aforementioned algebraic mapping operators, we implemented an algebraic mapping *translator*, and a proof-of-concept *engine*. The translator translates mapping rules in different mapping languages to a uniform mapping plan consisting of the algebraic mapping operators, while the proof of concept engine executes the mapping plan to generate RDF statements from heterogeneous data sources. Since our mapping algebra extends the SPARQL algebra (and thus naturally aligns with query-based mapping languages), we choose RML and ShExML from the categories of *dedicated mapping languages*, and *constraint-based languages*, respectively, to implement our translation algorithms described in Section 5.1 and Section 5.2. We specifically generate the mapping plan for the following versions: i) RML v1.1.1 [3][8], and ii) ShExML [6] from 2020 (ShExML v2020).

We choose RML v1.1.1 as RML is the prevalent declarative mapping language [8] – as evidenced by its ongoing support via the W3C Knowledge Graph Construction Community Group[9] – and the v1.1.1 version is mature with large implementation coverage[10]. The more recent version of RML [7] is backwards compatible with the previous version[11], hence, we do not expect breaking changes. We choose ShExML v2020 as it is independent of the other major mapping language families used in this work

This selection thus shows that our mapping algebra covers the semantics of at least one mapping language from the 3 categories of mapping languages mentioned in Section 2: RML-based languages are represented by RML v1.1.1 to compare against ShExML, and SPARQL-based languages are represented by the algebra definitions on which we based our algebraic operators on. This way, we ensure that our approach is not dependent upon a single mapping language.

Our proof-of-concept is used to demonstrate current coverage and practical feasibility of our proposed mapping algebra, but is not exhaustive in its current form, specifically concerning the ShExML translation. Syntactic sugar

[8] https://rml.io/specs/rml/v/1.1.1/
[9] https://www.w3.org/community/kg-construct/
[10] https://rml.io/implementation-report/
[11] https://kg-construct.github.io/rml-resources/portal/backwards-compatibility.html

such as *Query* declarations are not supported since it is used to make the ShExML document more human-readable and do not add or change mapping steps. We currently only support one ShExML transformation operation (string concatenation), and no joins. We do not support the ShExML v2020 join – currently known as *substitution*[12] – since it was defined with the usage of both the *UNION* and the *JOIN* keywords without detailed clarification on the operational semantics [6].

The algebraic mapping translator is implemented in Rust[13] and utilizes Sophia [36] as a library for handling RDF types. The translator[14] is called "Algebraic Mapping Loom: Weaving Mapping Languages" ("AlgeMapLoom", v0.4.0), and contains different modules to implement a mapping language translator. The decision to use Rust as the implementation language is to leverage Rust's cross compilation capabilities to enable the usage of the translator code on multiple operating systems. Furthermore, Rust has a rich ecosystem of libraries to support the generation of bindings for multiple programming languages, enabling developers to use the translator engine's API from within their code. The output of the AlgeMapLoom translator is a graph data structure that is compliant with the Graphviz[15] format in DOT language [16]. For ease of implementing the algebraic mapping engine, we also provide a JSON output of the translated mapping plan together with the JSON schema [17].

The proof-of-concept algebraic mapping engine[18] is implemented in JavaScript, called "RMLWeaver-JS" (v0.1.1), to show that the translated mapping plan is mapping and programming language agnostic. For this work, we only support processing CSV files[19] with RMLWeaver-JS. As the mapping plan is source-independent – except for the extensible source operator – only supporting CSV files is sufficient to prove the working of our proof-of-concept. We employed the reactive programming paradigm[20] when implementing RMLWeaver-JS to ensure that input data is processed in a streaming manner, resulting in lower memory usage. The following sections give an overview of the respective algorithms used to translate RML and ShExML into a mapping plan consisting of algebraic mapping operators.

5.1. RML translation

RML v1.1.1 describes how triples maps are used to generate RDF statements. A triples map is linked to a source – called *logical source* in RML – which provides the necessary data to generate the RDF statements described by the triples map. Multiple triples maps can have the same RML *logical source*. For translating a logical source to a *Source* operator (Section 4.2), we extract the iterators and fields related to a logical source from the triples map, as described in the work on RML Fields [29].

Then, all *triples maps* are grouped according to their associated *Source* operators. The grouped triples maps are then iterated over individually and the associated *term maps*[21] are translated into corresponding *Projection*, *Join*, *Extend*, *Serialize* operators as follows. For each term maps (subject, predicate, and object maps), the *references*[22] are extracted as *projection attributes* to create projection operators. The created *Projection* operators are applied directly after the previously created *Source* operators. This results in a *partially projected* mapping plan, used throughout the rest of the algorithm to build the final mapping plan representing the mapping process described by the RML document.

Once the *Projection* operators are created, the triples maps are partitioned into groups with and without referencing object map[23] to be further translated into sub-mapping plans *with* and *without* joins.

[12] https://shexml.herminiogarcia.com/spec/#substitution
[13] https://www.rust-lang.org/
[14] https://github.com/RMLio/algemaploom-rs/releases/tag/v0.4.0
[15] https://graphviz.org/
[16] https://graphviz.org/doc/info/lang.html
[17] https://rmlio.github.io/algemaploom-rs/schema/schema_doc.html
[18] https://github.com/RMLio/rmlweaver-js/releases/tag/v0.1.1
[19] With some limitations in error handling for malformed CSV records and handling deduplication.
[20] https://rxjs.dev/
[21] RML term maps: https://rml.io/specs/rml/#term-map
[22] RML reference: https://rml.io/specs/rml/#reference
[23] Reference object map: https://rml.io/specs/rml/#logical-join

For triples map with referencing object maps, a *Fragmenter* operator and a *Join* operator is created to join the partially projected mapping plan involving the *child* and *parent* triples map for each referencing object map associated with the triples map. The *Fragmenter* operator is created with the fragment mapping, $(f_{default}, f_{join})$, to broadcast the output of the previous operator to go to both the *Join* operator and the other downstream operator not involved in the join operation. If multiple *Join* operators must be created, the previously created *Fragmenter* operator is updated with new fragment mappings to broadcast the output to more operators. The type of the *Join* operator is determined by the presence of *join conditions*[24]. If the join condition exists, a *θ-join* operator is created using the attributes specified by the child and the parent references. Otherwise, a *natural join* operator is created. For triples map without referencing object maps, the aforementioned step for the *Join* operator creation is skipped.

Then, information about the term maps are utilized to create *extend expressions*. For example, a constant-valued term map with the term type IRI is translated to generate a nested *extend expression* (Definition 7) where an IRI data typing extend expression is applied to the return value of the constant value generating extend expression (e.g. *irify(constant(value))*). The new variable, v_{new}, to which the corresponding extend expression is bound to, is generated uniquely for each term map. The *Extend* operator is generated from the aforementioned pairs of variables and extend expressions, and applied after the previously created operator (i.e. either a *Join* or a *Projection* operator, depending on the presence of the referencing object maps).

Finally, the *Serialize* operator is created based on the combination of term maps for the triples map. Subject, predicate, object, and graph maps are used to generate *quad patterns*[25] with variables for each term.

The proposed RML *logical target* [33] is partially supported: only the default logical target is interpreted by creating a default *Target* operator that pipes the generated RDF quads to the terminal's standard output, for all the mapping tuples having the *default fragment*.

5.2. ShExML translation

Unlike RML, ShExML documents[26] have a structure split into two blocks: i) *declarations* and ii) *generators*. The *declarations* block contains individual lines defining *sources*, *iterators*, *prefixes*, and *expressions*. The declarations have the following structure: <*type*> <*variable*> <*statement*>. Each declaration is aliased with a *variable* which can be used within other declarations - introducing interdependency between different declarations. Thus, it is important to group related declarations together to generate our mapping plan. The *generators* block contains *shapes* and *graphs*, which in turn can contain nested shapes. The syntax for defining the graphs and shapes are the same with the ShEx specifications[27] with some modifications by ShExML.

First, unique combinations of the *source* and the *iterators* variables, used inside the ShExML *expressions*' statements, are used to generate the algebraic *Source* operators. One *Source* operator is generated for each unique combination of a source and an iterator variables. The generated *Source* operator is grouped with the variables of the expression definitions, which reference the same source as the *Source* operator. This results in pairs where every *Source* operator is paired with a set of expression variables. We shall annotate the set of expression variables as V_{expr} which will be referred to in the remaining algorithm steps.

For each pair of *Source* operator with V_{expr}, we perform a two-step processing: i) generation of the RDF quad patterns that could be generated for the current *Source* operator, and ii) generation of the relevant algebraic mapping operators.

RDF quad patterns are derived from the shapes and graphs in the *generator* block of ShExML document. These RDF quad patterns also contain metadata such as RDF data type or term type (IRI, Blank node, Literal) to aid in the generation of the value to be bound to the variables in the RDF quad pattern. RDF quad patterns are generated, and added to the set of RDF quad patterns, if the subject node and the object node of a ShExML shape references one of the expression variables $v \in V_{expr}$. Furthermore, string templating extend expressions are generated for both the subject and the object nodes since the terms for these nodes need to be generated dynamically during the

[24] RML join conditions: https://rml.io/specs/rml/#join-condition
[25] Quad Pattern: https://www.w3.org/TR/sparql11-query/#rQuadPattern
[26] https://github.com/herminiogg/ShExML
[27] https://shex.io/shex-semantics/

mapping process. The generated string templating extend expressions will be used later on for the configuration of the *Extend* operator. Predicate nodes in ShExML are predefined as a constant IRI. If there is a linking shape[28], and the expression variable used in the subject node of the nested shape is $v_{subj} \in V_{expr}$, an RDF quad pattern is generated where the object term variable is the same as the subject term variable of the nested shape, and it is added to the set of RDF quad patterns. The generated set containing the RDF quad patterns are used later for the creation of the *Serialize* operator (Section 4.8).

Once the RDF quad patterns are generated, we generate the relevant algebraic mapping operators based on the type of *expression declarations* with expression variable $v \in V_{expr}$. If the expression declarations are transformations, such as string operation[29], the relevant built-in extend expressions (Definition 7) are created. The extend expressions are paired with the ShExML expression variable to which the generated value will be bound. These pairs of variables and extend expressions are used to create an *Extend* operator which is applied on the *Source* operator currently being processed. This step is optional depending on the presence of the transformation expressions.

Basic expressions[30] are translated after the transformation expressions. Basic expressions in ShExML behave just like a *Rename* operator (Section 4.5), where the values generated from the <*statement*> are aliased with the associated expression variable. Thus, a *Rename* operator is generated, with the rename pairs derived from the basic expressions whose $v \in V_{expr}$. The *Rename* operator is applied directly as the next step of the mapping plan. This step can also be optional depending on the presence of basic expressions.

Afterwards, the extend expressions (Definition 7) for type casting, derived from the metadata of the generated RDF quad patterns, are generated and paired with the corresponding variable in the RDF quad pattern. For example, an IRI type casting statement is generated for a subject node and paired with the variable of the subject node in the RDF quad pattern. The generated pairs of variables and extend expressions are used to create an *Extend* operator, with typecasting extend expressions. The created *Extend* operator is applied after the previous step.

Finally, the previously generated RDF quad patterns are used to generate the *Serialize* operator. Since ShExML can not specify targets, the default *Target* operator is created to pipe the generated RDF quads to the terminal's standard output.

5.3. Mapping plan

Once we apply the aforementioned algorithms to translate RML or ShExML documents, a mapping plan is generated. Figure 2 shows an example mapping plan generated from RML document (Listing 3) using the algorithm described in Section 5.1 whereas Figure 4 shows the mapping plan generated from the ShExML document (Listing 5).

5.3.1. RML translation result

When translating a simple RML document (Listing 3), the *Projection* operator projects the attributes referenced by the term maps in RML. The projected attribute is *name*, which is referenced in the subject map's template and the object map's reference respectively. The *Extend* operator contains pairs of attributes and *extend expressions* for the generation of the terms required for the mapping process. In Figure 2, the *Extend* operator binds the variable $?tm0\_sm$ to the RDF generated by evaluating the extend expression $Iri(Template(\text{"http://example.com/\{name\}"}, \text{"name"} \rightarrow Reference(\text{"name"})))$ resulting in an IRI with the string template, "http://example.com/{name}", where the value for the variable "name" is retrieved from the current solution mapping being operated upon by the *Extend* operator. The quad pattern, used by the *Serialize* operator for serializing the data into N-Triples, is derived from the usage of the subject and predicate object maps in the RML document. As there are no logical targets specified in the example RML document, the default *Target* operator is used: the generated N-Triples are piped to the standard output of the terminal.

Translating the RML document in Listing 3 does not produce the *Fragmenter* nor the binary operators such as *θ-join* operator. These operators are only present when the RML document describes a join operation between two

[28] ShExML linking shapes: https://shexml.herminiogarcia.com/spec/#linking-shapes
[29] https://shexml.herminiogarcia.com/spec/#string-operation
[30] https://shexml.herminiogarcia.com/spec/#basic-expression

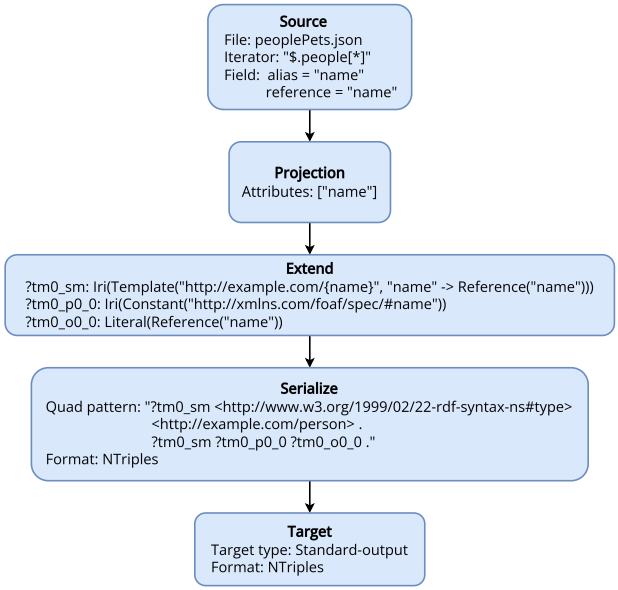

Figure 2. A simple mapping plan generated from the mapping process described by the RML document in Listing 3

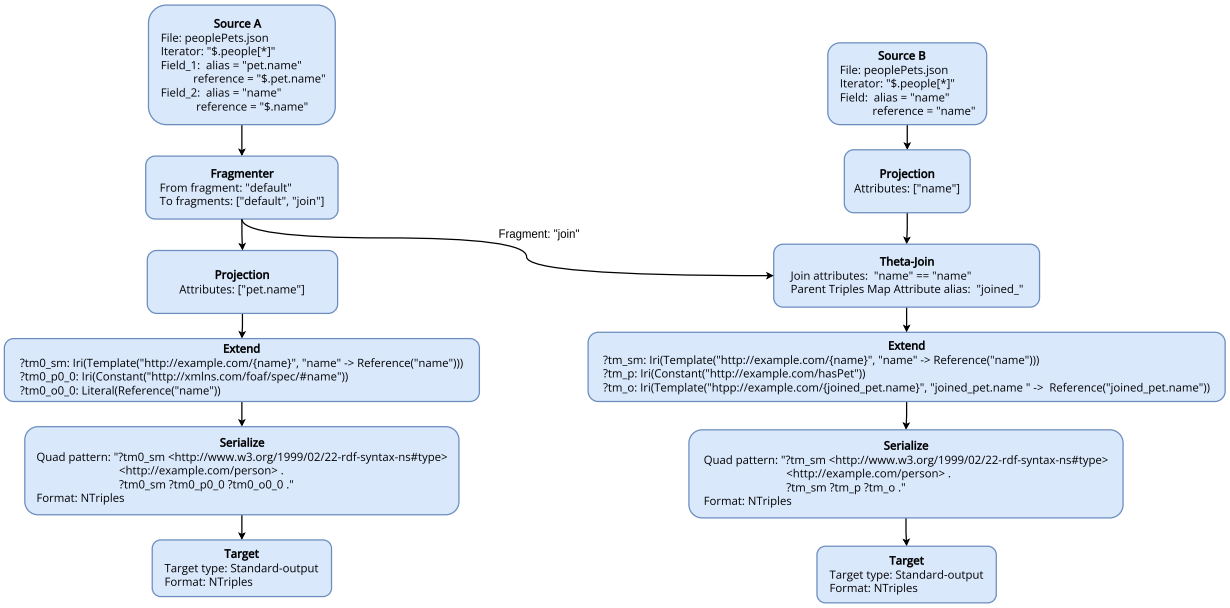

Figure 3. A mapping plan with joins generated from the RML document in Listing 4

triples map such as the example RML document in Listing 4. The mapping plan generated from the RML document in Listing 4 contains a *Fragmenter* operator to fragment the mapping tuples to two downstream operator; the *Projection* operator, and the *θ-join* operator. The *Projection* operator after the *Fragmenter* operator, in the downstream path from Source A, ensures that the mapping tuples do not contain attributes not required by the rest of the downstream operators.

5.3.2. *ShExML translation result*

There are two major differences between the mapping plan generated by RML translation and the ShExML translation. First, ShExML translation generates a *Rename* operator as part of the mapping plan due to the basic

Listing 3: Example RML document, with 1 subject and 1 predicate-object map, processing data from a JSON file. The input data is the JSON data shown in Listing 1.

```
1  @prefix rr: <http://www.w3.org/ns/r2rml#> .
2  @prefix rml: <http://semweb.mmlab.be/ns/rml#> .
3  @prefix ql: <http://semweb.mmlab.be/ns/ql#> .
4  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
5  @base <http://example.com/base/> .
6
7  <TriplesMap1>
8      a rr:TriplesMap;
9      rml:logicalSource [
10     rml:source "peoplePets.json";
11     rml:referenceFormulation ql:JSONPath;
12     rml:iterator "$.people[*]"
13   ];
14
15   rr:subjectMap [
16     rml:template "http://example.com/{name}";
17     rr:class <http://example.com/person>;
18   ];
19
20   rr:predicateObjectMap [
21     rr:predicate foaf:name;
22     rr:objectMap [ rml:reference "name" ];
23   ].
```

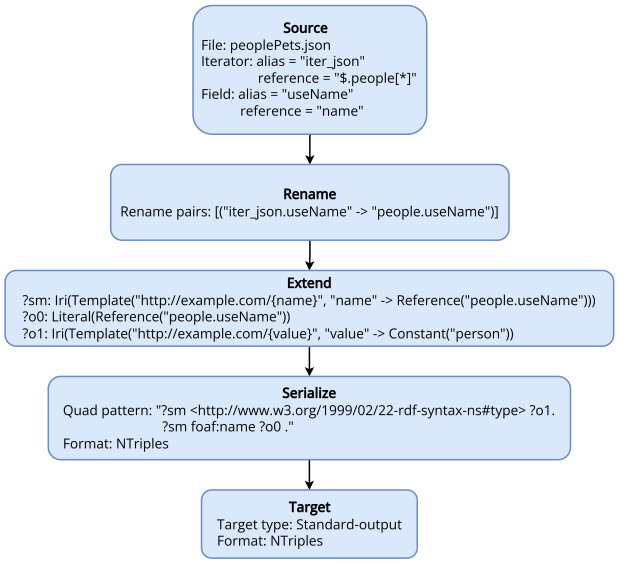

Figure 4. A mapping plan generated from the ShExML document in Listing 5

expression on line 12 in Listing 5. Lastly, the *Extend* operator has an extra variable binding for the generation of a constant IRI term namely *<http://example.com/person>*. This arises from the ShExML translation algorithm (Section 5.2) which generates extend expressions for both the subject and the object nodes in ShExML's shapes and graphs block. The other operators are semantically similar to those generated from the RML document.

6. Evaluation

This work introduces the definitions of algebraic mapping operators. We conduct an *empirical evaluation* of the algebraic mapping operators, using the aforementioned RML and ShExML mapping languages (Section 5). We implemented a reference algebraic mapping engine for the evaluation. Two types of empirical evaluation are carried

Listing 4: Example RML document where two triples maps are joined based on the attribute *"name"*. The input data is the JSON data shown in Listing 1.

```
@prefix rr: <http://www.w3.org/ns/r2rml#> .
@prefix rml: <http://semweb.mmlab.be/ns/rml#> .
@prefix ql: <http://semweb.mmlab.be/ns/ql#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/spec/#>.
@prefix ex: <http://example.com/> .
@base <http://example.com/base/> .

<TriplesMap1>
    a rr:TriplesMap;
    rml:logicalSource [
    rml:source "peoplePets.json";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.people[*]"
  ];

  rr:subjectMap [
    rr:template "http://example.com/{name}";
    rr:class <http://example.com/person>;
  ];

  rr:predicateObjectMap [
    rr:predicate ex:hasPet;
    rr:objectMap [
        rr:parentTriplesMap <TriplesMap2>;
        rr:joinCondition [
            rr:child "name";
            rr:parent "name";
        ];
    ];
  ].

<TriplesMap2>
    a rr:TriplesMap;
    rml:logicalSource [
    rml:source "peoplePets.json";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.people[*]"
  ];

  rr:subjectMap [
    rr:template "http://example.com/{pet.name}";
    rr:class <http://example.com/pet>;
  ];

  rr:predicateObjectMap [
    rr:predicate foaf:name;
    rr:objectMap [ rml:reference "pet.name" ];
  ].
```

out for this work: i) completeness of the defined algebraic mapping operators, and ii) the impact on performance of a mapping engine utilizing algebraic mapping operators. The first evaluation shows that this work is sufficient to create complete mapping engines. The second evaluation shows that utilizing algebraic mapping operators results in performance of real-world mapping engines comparable to the state of the art. Figure 5 shows an example execution pipeline consisting of AlgeMapLoom-rs and RMLWeaver-JS used to translate and execute RML document. The same pipeline setup is also used for ShExML evaluation.

6.1. Completeness of algebraic mapping operators

Evaluating the completeness of the algebraic mapping operators is done by translating test cases, provided by the RML and ShExML reference implementations, into a mapping plan using the defined algebraic mapping operators. We then execute the generated mapping plan with our reference implementation, RMLWeaver-JS, and check the output of our implementation against the output of the reference implementations of RML and ShExML.

For RML, we use the RML v1.1.1 specification conformance test cases. Since RMLWeaver-JS only supports processing CSV files, we only chose the test cases using CSV files as input data source. The test cases

Listing 5: Example ShExML document equivalent to the RML document in Listing 3.

```
PREFIX rr: <http://www.w3.org/ns/r2rml#>
PREFIX rml: <http://semweb.mmlab.be/ns/rml#>
PREFIX ql: <http://semweb.mmlab.be/ns/ql#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/spec/#>
PREFIX : <http://example.com/>

SOURCE peoplePetsJSONFile <peoplePets.json>
ITERATOR iter_json <jsonpath: $.people[*]> {
    FIELD useName <name>
}
EXPRESSION people <peoplePetsJSONFile.iter_json>

:Films :[people.useName] {
    a :person ;
    foaf:name [people.useName] ;
}
```

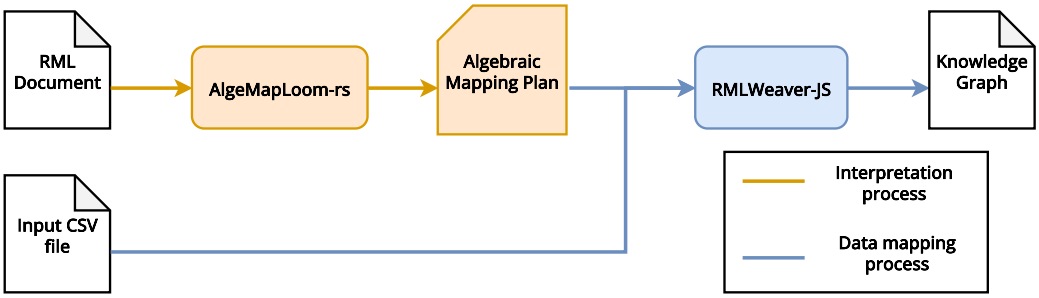

Figure 5. Algebraic mapping engine pipeline where an RML document is first translated into an algebraic mapping plan which is used to generate the KG.

for RML are available at the RMLWeaver-JS repository: https://github.com/RMLio/rmlweaver-js/tree/v0.1.1/test/rml-mapper-test-cases-csv.

For ShExML, the test cases provided with the reference implementation utilize heterogeneous data formats and sources such as a mix of JSON and XML or SPARQL endpoints. Since RMLWeaver-JS only supports input data files in CSV format, we adapted the test cases to utilize only CSV files as input. ShExML reference implementation's test cases evaluates multiple features of the ShExML language per test case. Therefore, we also split up the existing test cases into multiple smaller test cases to conduct a more granular evaluation of RMLWeaver-JS's execution of ShExML documents. For example, ShExML test case called *MultipleElementTest* tests for the usage of both multiple *Iterators*, and multiple *Basic Expressions* statements. We split it into two smaller test cases which evaluates the usage of multiple iterators and multiple basic expressions statements separately. This resulting set of test cases for ShExML are available at the RMLWeaver-JS repository: https://github.com/RMLio/rmlweaver-js/tree/v0.1.1/test/shexml.

6.1.1. Results and discussion

RMLWeaver-JS produces the same output as the reference RML implementation[31] for all 39 out of the 39 RML CSV test cases (100%), covering 100% of the operational semantics of the RML CSV test cases (Table 12).

For the ShExML test cases, RMLWeaver-JS generates the same output as the reference v0.5.1 ShExML implementation[32].

Table 12 shows the number of features supported by RMLWeaver-JS in execution for both RML v1.1.1 and ShExML v2020. RMLWeaver-JS supports 16 out of 23 (65%) of ShExML features, and supports 11 out of 11

[31] https://github.com/RMLio/rmlmapper-java/releases/tag/v7.0.0, https://doi.org/10.5281/zenodo.11518178
[32] https://github.com/herminiogg/ShExML/releases/tag/v0.5.1

Table 12

Our solution supports 65% of ShExML v2020 features and 100% of RML v1.1.1 features. The features are aligned across the two languages in terms of their functionality in mapping heterogeneous data to RDF. For example, *string operations* in ShExML is equivalent to the usage of *template-valued term maps* in RML.

ShExML v2020 feature	Is supported	RML v1.1.1 feature	Is supported
Declarations	7/8		2/2
Prefix	✓		
Source	✓	Logical source	✓
Query			
Iterator	✓	Logical iterator	✓
Nested Iterator	✓		
Fields	✓		
Push Fields	✓		
Pop Fields	✓		
Expressions	3/7		2/2
Basic	✓		
Union	✓		
String Operation	✓	Template-valued term maps	✓
Join		Referencing object map + join condition	✓
Matcher			
Autoincrement			
Dynamic Function			
Shapes & Graphs	6/8		7/7
Basic	✓	Reference-valued term maps	✓
Basic (constant)	✓	Constant-valued term maps	✓
Link shapes	✓	Referencing object map	✓
Matcher			
Datatypes static + dynamic	✓	Object map + datatype	✓
Langtype static + dynamic	✓	Object map + language tag	✓
Conditional + dynamic functions			
		Predicate map	✓
Graphs	✓	Graph map	✓

(100%) of RML v1.1.1 features. The exhaustive alignment of all ShExML features (such as Joins) to the introduced mapping algebra is future work. All testing code and results are published on GitHub[33].

6.2. *Performance of an algebraic mapping engine*

To show that the implementation of an algebraic mapping engine does not have a large negative impact on the performance, we participated [37] in the first part of the performance track of the 2024 Knowledge Graph Construction Challenge[34]. We highlight results of that participation in this paper, extend on its descriptions, put it in context with the other participating mapping engines, and align the results with learnings concerning our proposed mapping algebra.

The performance track's first part evaluates the mapping engines' performance when handling diverse knowledge graph construction parameters with synthetic datasets, using RML mapping rules. Mapping engines are evaluated by changing the following properties of the data: the number of data records, data properties, duplicates, empty values, and input files. It also changes the following properties of the mapping rules: the number of subjects, predicates

[33] https://github.com/RMLio/rmlweaver-js/blob/v0.1.1/test/rml_tests.js
[34] https://kg-construct.github.io/workshop/2024

Table 13

FlexRML is the most performant engine when constructing knowledge graph from varying triples maps and predicate-object maps. It performs best when the number of triples maps (TM) is closer to the number of CPU cores on the machine.

Test cases	Engines	Execution time (s)	CPU usage (s)	Peak RAM (GB)
1TM 15POM	Mapping-template	-	-	-
	FlexRML	**6.54**	**6.81**	**0.47**
	RMLWeaver-JS	11.26	13.21	0.54
	RPT-Sansa	43.25	133.92	4.50
	RMLStreamer	44.76	113.01	6.10
3TM 5POM	Mapping-template	-	-	-
	FlexRML	**3.79**	**9.59**	**0.51**
	RMLWeaver-JS	15.66	17.73	0.55
	RPT-Sansa	44.18	122.37	4.40
	RMLStreamer	43.52	116.28	6.06
15TM 1POM	Mapping-template	-	-	-
	FlexRML	**6.34**	**18.02**	**0.46**
	RMLWeaver-JS	42.57	46.65	0.55
	RPT-Sansa	48.68	99.80	3.99
	RMLStreamer	40.74	108.58	6.09

and objects, and finally, the number and type of joins used. For the measurements, the following metrics of the participating mapping engines are measured: i) maximum RAM usage (GB), ii) CPU usage (s), and iii) execution time (s). The interpretation of CPU usage is as follows: 100% CPU usage is achieved when the CPU usage time (in seconds) equals the product of the execution time and the number of CPU cores available on the machine. In our evaluation setup, 100% CPU usage is four times the execution time as our machine has 4 cores available.

There are 5 mapping engines, including RMLWeaver-JS, participating in the performance track of the challenge. The engines can be classified into two major groups: those based on data processing frameworks (e.g. Apache Spark and Flink), and those without using data processing frameworks. RPT-Sansa [38], Mapping-template [39], and RMLStreamer [13] are based on Apache Spark[35], Apache Velocity[36], and Apache Flink[37] data processing frameworks respectively. The other 2 engines are FlexRML [40, 41], implemented in C++, and our JavaScript implementation RMLWeaver-JS.

All the engines are evaluated on a virtual machine provided by the organizers, which has a standardized specification to ensure a fair evaluation. Each engine is provided with its own separate virtual machine for evaluation. The virtual machine has a 64 bit architecture, and it is configured with an Intel(R) Xeon(R) Gold 6161 CPU at 2.20GHz with 4 cores, 16765 MB of RAM memory, and 150 GB of storage space. The operating system of the machine is Ubuntu 22.04.03 LTS. The execution of the experiment is done using the tools provided by the challenge organizers [42], isolated via Docker container[38].

Since the results are significantly more verbose than the completeness evaluation in Section 6.1, we present our results and discuss their causes separately.

6.2.1. Results

The full results of the challenge for knowledge graph parameters can be found on Zenodo[39] by downloading the file *System-Results-Challenge-2024.zip*. These results are based on the submissions by the authors of their respective engines. Thus, we can not conclude whether the engine failed at executing the test case or the results are omitted

[35] https://spark.apache.org/
[36] https://velocity.apache.org/
[37] https://flink.apache.org/
[38] Docker: https://www.docker.com/
[39] https://zenodo.org/doi/10.5281/zenodo.10721874

Table 14
Number of properties in CSV data records has little to no impact on the memory usage of RMLWeaver-JS.

Test cases	Engines	Execution time (s)	CPU usage (s)	Peak RAM (GB)
1 column	Mapping-template	5.71	**1.73**	2.10
	FlexRML	**5.63**	5.80	**0.42**
	RMLWeaver-JS	17.78	19.52	0.47
	RPT-Sansa	40.12	86.10	2.82
	RMLStreamer	37.11	93.08	6.09
10 columns	Mapping-template	**15.26**	**3.73**	3.22
	FlexRML	43.66	43.82	0.79
	RMLWeaver-JS	65.63	69.78	**0.52**
	RPT-Sansa	100.24	351.04	11.14
	RMLStreamer	168.16	425.24	6.11
30 columns	Mapping-template	**39.34**	**7.80**	5.22
	FlexRML	137.30	140.21	1.72
	RMLWeaver-JS	172.80	194.09	**0.50**
	RPT-Sansa	319.38	1203.27	5.37
	RMLStreamer	462.99	1311.78	6.15

from the list. We skip the presentation of the results for duplicates and empty values test cases, since RMLWeaver-JS does not deduplicate the generated triples nor handle empty values in the columns by ignoring them. We discuss the results of the other test cases and compare our performance against the other participants in the following paragraphs.

Table 13 shows the engines' performance when the number of triples map (TM) and predicate-object maps (POM) in the RML document changes, while the input dataset size stays the same. For the other engines, there is no difference in CPU usage and execution time increase when compared to RMLWeaver-JS. Where for FlexRML execution time shortens and CPU usage maximizes for the test case with 3 TM and 5 POM, for RMLWeaver-JS execution time and CPU usage increases across the test cases. It even has a 4-fold increase for execution time and CPU usage for the test case with 15 TM and 1 POM.

Table 14 shows the results of the test cases to evaluate the performance impact by increasing the number of columns in the input CSV dataset. RMLStreamer and RMLWeaver-JS maintain constant memory usage, while for the other engines memory usage increases with the number of columns. RMLWeaver-JS maintains the lowest memory usage amongst engines for the 10 and 30 columns CSV data records test cases. For all engines, execution time and CPU usage increases with the number of columns. Mapping-template is the fastest in terms of execution time and lowest CPU usage. FlexRML is only slightly faster, by 0.08 seconds, than Mapping-template for the first test case with 1 column CSV data records.

Table 15 contains the results of the test case with increasing number of CSV data records. For the test case with 10K rows, FlexRML is the fastest engine (1.23 seconds) using the lowest memory (0.4 GB) while Mapping-template's CPU usage is the lowest at 0.55 seconds. Once the number of rows reaches 100k, Mapping-template becomes faster than FlexRML and uses noticeably less CPU time (8 times less than FlexRML), but memory usage increases with 0.44 GB compared to the 10k rows test case. However, Mapping-template fails to produce any results for 10M rows of CSV data. Throughout the experiment, RMLWeaver-JS maintains a constant memory usage of approximately 0.5 GB even for the 10M rows test case. RMLStreamer uses the same amount of memory of around 6.1 GB for both 100k and 10M rows of CSV data. This is similar to the amount of memory RMLStreamer uses for the test cases in Table 14.

Table 16 provides the measurements of evaluating on the join related test cases where 100% of the data records are eligible to be joined. All engines maintain similar performance across test cases Join N-M based on $min(N, M)$. For example, FlexRML has similar performance for test cases Join 5-5 and Join 5-10 across all metrics measured. Furthermore, FlexRML is the fastest engine across all join test cases with the lowest memory and CPU usage.

Table 15

RMLWeaver-JS manages to keep memory usage constant around 0.5 GB with increasing input data size, while FlexRML is the fastest.

Test cases	Engines	Execution time (s)	CPU usage (s)	Peak RAM (GB)
10K rows	Mapping-template	2.41	**0.55**	1.04
	FlexRML	**1.23**	1.33	**0.40**
	RMLWeaver-JS	2.56	3.34	0.48
	RPT-Sansa	33.06	97.43	1.68
	RMLStreamer	23.49	51.06	1.75
100K rows	Mapping-template	**4.95**	**1.16**	1.48
	FlexRML	8.28	8.40	**0.46**
	RMLWeaver-JS	13.51	14.76	0.49
	RPT-Sansa	48.35	152.83	5.12
	RMLStreamer	49.84	129.47	6.16
10M rows	Mapping-template	-	-	-
	FlexRML	**943.34**	**963.50**	11.82
	RMLWeaver-JS	1116.40	1249.97	**0.54**
	RPT-Sansa	1569.04	5909.89	6.75
	RMLStreamer	1768.58	6918.19	6.17

Table 16

Performance of the mapping engines, on a test case Join N-M, depends on $min(N, M)$. 100% of the data records are eligible to be joined in the test cases presented.

Test cases	Engines	Execution time (s)	CPU usage (s)	Peak RAM (GB)
Join 10-1	Mapping-template	-	-	-
	FlexRML	**15.35**	**19.57**	**0.56**
	RMLWeaver-JS	30.03	33.75	0.64
	RPT-Sansa	40.41	121.75	5.11
	RMLStreamer	66.98	222.97	6.36
Join 1-10	Mapping-template	-	-	-
	FlexRML	**15.03**	**19.14**	**0.50**
	RMLWeaver-JS	30.04	33.60	0.64
	RPT-Sansa	38.97	119.19	4.13
	RMLStreamer	60.14	195.16	6.36
Join 5-5	Mapping-template	-	-	-
	FlexRML	**23.41**	**32.29**	**0.59**
	RMLWeaver-JS	82.04	90.99	0.81
	RPT-Sansa	50.70	149.82	5.32
	RMLStreamer	109.65	403.17	6.36
Join 5-10	Mapping-template	-	-	-
	FlexRML	**22.51**	**31.44**	**0.59**
	RMLWeaver-JS	81.64	90.94	0.79
	RPT-Sansa	47.73	149.37	4.83
	RMLStreamer	120.97	421.70	6.35

6.2.2. Discussion

Analysing the results presented in Section 6.2.1 reveals several potential improvements for implementing a more efficient algebraic mapping engine.

Compared to the other engines, RMLWeaver-JS exhibits an abnormal behaviour for the test cases where the number of TMs and POMs changes inversely in the RML document (Table 13). For example, RMLWeaver-JS is the only engine with a significant spike in execution time from *11.26 seconds* to *42.57 seconds* for the test cases 1TM 15POm, and 15TM 1POm respectively. This can be explained due to the manner in which AlgeMapLoom (Section 5) translates the RML document into the algebraic mapping plan used by RMLWeaver-JS. The test cases include multiple TMs, each with its own definition of a logical source, all referring to the same CSV data file and using the same iterator. Due to the lack of detection for semantically similar data sources, the interpreter generates a distinct source operator for each logical source definition identified. In the specific test case of 15 TMs and 1 POM, a total of 15 source operators are generated, with each TM associated with one of these source operators. RMLWeaver-JS is implemented in JavaScript without worker threads, thus inherently single-threaded and can only execute one task at a time. Thus, processing the CSV data file 15 times instead of just once is done sequentially. This results in a noticeable increase in both execution time and CPU usage.

For the test cases regarding increasing number of properties and records, RMLWeaver-JS and RMLStreamer managed to maintain constant memory even if the input data columns or rows increase. This consistent memory usage is due to the way both engines process data. On one hand, RMLStreamer – built on the Apache Flink stream processing framework – processes input CSV rows one at a time. Consequently, RMLStreamer maintains a constant memory usage of around 6.1 GB, even for inputs with a higher number of columns and records. On the other hand, RMLWeaver-JS, implemented based on reactive programming paradigm, also processes the input CSV records one at a time, leading to the same constant memory usage. The substantial difference in memory usage, with RMLWeaver-JS using around 0.5 GB and RMLStreamer using approximately 6 GB, is due to the overhead of the underlying Apache Flink framework. Apache Flink allocates a fixed amount of heap memory for the Java Virtual Machine on which RMLStreamer is executed. Thus, RMLWeaver-JS has a lower memory usage than RMLStreamer due to the implementation not relying on data processing frameworks.

As observed in Table 16, all engines demonstrate the same performance across two different test cases in Join N-M, depending on the $min(N, M)$. We attempt to provide an explanation for such behaviour for RMLWeaver-JS [37], however, the same conclusion cannot be extended to other engines since we lack the details of their implementations. The explanation is as follows. Join N-M is a test case containing two sources S_n and S_m, where there are N records from S_n eligible to be joined with M records from S_m. RMLWeaver-JS employs a simple hash-join algorithm to join data from two different sources. It creates two hash maps – one for each source – for bookkeeping when joining the CSV records. Assuming the data are going to be joined on an attribute A, and provided $M < N$ with M records coming from S_m and N records coming from S_n. In order for RMLWeaver-JS to achieve the same performance as presented in this work for the joins, M records from S_m needs to arrive first at the join operator and be stored in the hash map $HashMap_{S_m}(A)$. This ensures that the amortized cost of joining the N records from S_n is lower since it only requires $HashMap_{S_m}(A)$ to be looped through M times, where $M < N$. Otherwise, the amortized cost will be higher if N records from S_n arrives first, causing the $HashMap_{S_n}(A)$ to be looped through at least N times with $M < N$. The aforementioned explanation applies to both Join 5-5 and Join 10-5, where RMLWeaver-JS exhibits similar performance in terms of execution time, CPU usage, and memory usage.

Summarizing the results, RMLWeaver-JS achieved the second place for the Track 2 performance challenge in Knowledge Graph Construction Workshop, handing first place to FlexRML [42]. This achievement shows that it is possible to implement a performant algebraic mapping engine in JavaScript for web browsers, potentially empowering web clients and servers with knowledge graphs generated from heterogeneous data sources.

7. Conclusion

In this paper, we presented a mapping language-independent mapping algebra consisting of algebraic mapping operators *Source*, *Projection*, *Extend*, *Rename*, *Fragmenter*, *Natural join*, *θ-join*, *Left outer-join*, *Union*, *Serialize*, and *Target*. We empirically showed how this mapping algebra provides (partial) operational semantics by translating

mapping rules of two existing but very different mapping languages (RML and ShExML, 100% and 63% of feature coverage, respectively), to a mapping plan consisting of our introduced algebraic mapping operators. We showed practical feasibility of our approach via a proof-of-concept algebraic mapping engine, RMLWeaver-JS, achieving second place in the Knowledge Graph Construction Workshop's performance challenge.

For future work, we will exploit this mapping algebra for theoretical research in mapping processes: translating mapping rules – in multiple mapping languages – into mapping plans conforming to our mapping algebra opens the door to static analysis of mapping rules, e.g. for verification and optimization. After further completion of our proof-of-concept implementations, e.g. researching the alignment of the join semantics in ShExML with our mapping algebra, we will investigate a *mapping plan optimizer* to improve the mapping process regardless of the mapping language used. We will start by considering existing optimizations, such as mapping partitions [9] and mapping assertions [15], and present and benchmark them in a unified mapping algebra model.

Furthermore, the mapping algebra defined in this work could be used to develop a formal mapping language with a formal grammar. Such a language will benefit from the operational semantics already defined by the algebra. This will require further formalization of the mapping plan, for which the Algemaploom-rs output adhering to a specific JSON schema can serve as an inspiration, to develop the formal mapping language.

With the advent of this mapping algebra and algebraic mapping engines, users are no longer locked into using a specific mapping language for knowledge graph generation. The performance across languages will become consistent by having an algebraic mapping engine that is multilingual, hence mapping language design can focus on functionality, decoupled from performance.

References

[1] S. Sundara, S. Das and R. Cyganiak, R2RML: RDB to RDF Mapping Language, W3C Recommendation, W3C, 2012, https://www.w3.org/TR/2012/REC-r2rml-20120927/.

[2] C. Stadler, J. Unbehauen, P. Westphal, M.A. Sherif and J. Lehmann, Simplified RDB2RDF Mapping, in: *LDOW@WWW*, 2015. https://api.semanticscholar.org/CorpusID:18692672.

[3] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens and R. Van de Walle, RML: A Generic Language for Integrated RDF Mappings of Heterogeneous Data, in: *Proceedings of the 7th Workshop on Linked Data on the Web*, C. Bizer, T. Heath, S. Auer and T. Berners-Lee, eds, CEUR Workshop Proceedings, Vol. 1184, CEUR, 2014. ISSN 16130073. http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[4] B. Vu, J. Pujara and C.A. Knoblock, D-REPR: A Language for Describing and Mapping Diversely-Structured Data Sources to RDF, in: *Proceedings of the 10th International Conference on Knowledge Capture*, K-CAP '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 189–196. ISBN 9781450370080. doi:10.1145/3360901.3364449.

[5] M. Lefrançois, A. Zimmermann and N. Bakerally, A SPARQL Extension for Generating RDF from Heterogeneous Formats, in: *The Semantic Web 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 – June 1, 2017, Proceedings*, E. Blomqvist, D. Maynard, A. Gangemi, R. Hoekstra, P. Hitzler and O. Hartig, eds, Springer International Publishing, Portoroz, Slovenia, 2017, pp. 35–50. ISBN 978-3-319-58068-5. doi:10.1007/978-3-319-58068-5_3. http://www.maxime-lefrancois.info/docs/LefrancoisZimmermannBakerally-ESWC2017-Generate.pdf.

[6] H. García-González, I. Boneva, S. Staworko, J.E. Labra-Gayo and J.M.C. Lovelle, ShExML: improving the usability of heterogeneous data mapping languages for first-time users, *PeerJ Computer Science* **6** (2020), e318.

[7] A. Iglesias-Molina, D. Van Assche, J. Arenas-Guerrero, B. De Meester, C. Debruyne, S. Jozashoori, P. Maria, F. Michel, D. Chaves-Fraga and A. Dimou, The RML Ontology: A Community-Driven Modular Redesign After a Decade of Experience in Mapping Heterogeneous Data to RDF, in: *Proceedings of the International Semantic Web Conference (ISWC)*, Lecture Notes in Computer Science, Springer, Cham, 2023, pp. 152–175. ISSN 1611-3349. ISBN 9783031472435. doi:10.1007/978-3-031-47243-5$_9$.

[8] D. Van Assche, T. Delva, G. Haesendonck, P. Heyvaert, B. De Meester and A. Dimou, Declarative RDF graph generation from heterogeneous (semi-)structured data: A systematic literature review, *Journal of Web Semantics* (2022). doi:10.1016/j.websem.2022.100753.

[9] J. Arenas-Guerrero, D. Chaves-Fraga, J. Toledo, M.S. Pérez and O. Corcho, Morph-KGC: Scalable knowledge graph materialization with mapping partitions, *Semantic Web* (2022), 1–20. doi:10.3233/sw-223135.

[10] E. Iglesias, S. Jozashoori, D. Chaves-Fraga, D. Collarana and M.-E. Vidal, SDM-RDFizer: An RML Interpreter for the Efficient Creation of RDF Knowledge Graphs, in: *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, ACM, 2020. doi:10.1145/3340531.3412881.

[11] U. Simsek, E. Kärle and D.A. Fensel, RocketRML - A NodeJS Implementation of a Use Case Specific RML Mapper, *ArXiv* **abs/1903.04969** (2019). doi:10.48550/ARXIV.1903.04969.

[12] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh and A. Dimou, Parallel RDF generation from heterogeneous big data, in: *Proceedings of the International Workshop on Semantic Big Data - SBD '19*, S. Groppe and L. Gruenwald, eds, SBD '19, ACM Press, Amsterdam, Netherlands, 2019. ISBN 978-1-4503-6766-0. doi:10.1145/3323878.3325802. https://biblio.ugent.be/publication/8619808/file/8659668.pdf.

[13] Sitt Min Oo, G. Haesendonck, B. De Meester and A. Dimou, RMLStreamer-SISO: An RDF Stream Generator from Streaming Heterogeneous Data, in: *The Semantic Web – ISWC 2022*, U. Sattler, A. Hogan, M. Keet, V. Presutti, J.P.A. Almeida, H. Takeda, P. Monnin, G. Pirrò and C. d'Amato, eds, Springer International Publishing, Cham, 2022, pp. 697–713, Springer. ISBN 978-3-031-19433-7. doi:10.1007/978-3-031-19433-$7_4$0.

[14] A. Iglesias-Molina, A. Cimmino, E. Ruckhaus, D. Chaves-Fraga, R. García-Castro and O. Corcho, An ontological approach for representing declarative mapping languages, *Semantic Web* **15**(1) (2024), 191–221, Publisher: IOS Press. doi:10.3233/SW-223224. https://content.iospress.com/articles/semantic-web/sw223224.

[15] E. Iglesias, S. Jozashoori and M.-E. Vidal, Scaling up knowledge graph creation to large and heterogeneous data sources, *Journal of Web Semantics* **75** (2023). doi:10.1016/j.websem.2022.100755. http://arxiv.org/abs/2201.09694.

[16] N. Lopes, S. Bischof, S. Decker and A. Polleres, On the semantics of heterogeneous querying of relational, XML and RDF data with XSPARQL, in: *Proceedings of the 15th Portuguese Conference on Artificial Intelligence (EPIA 2011), Lisbon, Portugal*, Citeseer, 2011, pp. 10–13.

[17] L. Asprino, E. Daga, A. Gangemi and P. Mulholland, Knowledge graph construction with a façade: a unified method to access heterogeneous data sources on the web, *ACM Transactions on Internet Technology* **23**(1) (2023), 1–31.

[18] J. Pérez, M. Arenas and C. Gutierrez, Semantics and Complexity of SPARQL, *ACM Trans. Database Syst.* **34**(3) (2009). doi:10.1145/1567274.1567278.

[19] C. Bizer and A. Seaborne, D2RQ-treating non-RDF databases as virtual RDF graphs, in: *Proceedings of the 3rd international semantic web conference (ISWC2004)*, Vol. 2004, Springer Hiroshima, 2004.

[20] R. Cyganiak, Tarql: SPARQL for Tables, GitHub, 2012. https://github.com/tarql/tarql.

[21] S. Bischof, S. Decker, T. Krennwallner, N. Lopes and A. Polleres, Mapping between RDF and XML with XSPARQL, *Journal on Data Semantics* **1**(3) (2012), 147–185.

[22] E. Daga, L. Asprino, P. Mulholland and A. Gangemi, Facade-X: An Opinionated Approach to SPARQL Anything, in: *Further with Knowledge Graphs – Proceedings of the 17 th International Conference on Semantic Systems, 6–9 September 2021, Amsterdam, The Netherlands*, Studies on the Semantic Web, Vol. 53, IOS Press, 2021, pp. 58–73. ISSN 18681158, 22150870. doi:10.3233/SSW210035.

[23] F. Michel, L. Djimenou, C. Faron-Zucker and J. Montagnat, Translation of Heterogeneous Databases into RDF, and Application to the Construction of a SKOS Taxonomical Reference, in: *International Conference on Web Information Systems and Technologies*, Springer, 2015, pp. 275–296. doi:10.1007/978-3-319-30996-$5_1$4.

[24] A. Chortaras and G. Stamou, D2RML: Integrating Heterogeneous Data and Web Services into Custom RDF Graphs, in: *LDOW@WWW*, 2018. https://api.semanticscholar.org/CorpusID:51950275.

[25] E. Prud'hommeaux, I. Boneva, J.E. Labra Gayo and G. Kellogg, Shape Expressions Language 2.1, Draft Community Group Report, World Wide Web Consortium (W3C), 2018. http://shex.io/shex-semantics/.

[26] F. Priyatna, O. Corcho and J. Sequeda, Formalisation and experiences of R2RML-based SPARQL to SQL query translation using morph, in: *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 479–490–. ISBN 9781450327442. doi:10.1145/2566486.2567981.

[27] A. Chebotko, S. Lu and F. Fotouhi, Semantics preserving SPARQL-to-SQL translation, *Data & Knowledge Engineering* **68**(10) (2009), 973–1000. doi:https://doi.org/10.1016/j.datak.2009.04.001. https://www.sciencedirect.com/science/article/pii/S0169023X09000469.

[28] J. Unbehauen, C. Stadler and S. Auer, Optimizing SPARQL-to-SQL Rewriting, in: *Proceedings of International Conference on Information Integration and Web-Based Applications & Services*, IIWAS '13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 324–330–. ISBN 9781450321136. doi:10.1145/2539150.2539247.

[29] T. Delva, D.V. Assche, P. Heyvaert, B. Meester and A. Dimou, Integrating Nested Data into Knowledge Graphs with RML Fields, 2021. https://www.semanticscholar.org/paper/Integrating-Nested-Data-into-Knowledge-Graphs-with-Delva-Assche/cfd3929eb7eb98209acea307838be4c9ddc4d33c.

[30] A. Seaborne and S. Harris, SPARQL 1.1 Query Language, W3C Recommendation, W3C, 2013, https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[31] Min Oo, Sitt and De Meester, Ben and Taelman, Ruben and Colpaert, Pieter, Towards algebraic mapping operators for knowledge graph construction, 2023, p. 5. ISBN 978-3-031-47239-8.

[32] P.R. Halmos, *Naive Set Theory*, Undergraduate Texts in Mathematics, Springer New York, 1998. ISBN 9780387900926. https://books.google.be/books?id=x6cZBQ9qtgoC.

[33] D. Van Assche, G. Haesendonck, G. De Mulder, T. Delva, P. Heyvaert, B. De Meester and A. Dimou, Leveraging Web of Things W3C Recommendations for Knowledge Graphs Generation, in: *Web Engineering, 21st International Conference, ICWE 2021, Proceedings*, 2021, pp. 337–352. doi:10.1007/978-3-030-74296-6_26. https://dylanvanassche.be/assets/pdf/icwe2021-wot-logical-target.pdf.

[34] W.D. Blizard, Multiset Theory, *Notre Dame Journal of Formal Logic* **30**(1) (1988), 36–66. doi:10.1305/ndjfl/1093634995.

[35] S. Gössner, G. Normington and C. Bormann, JSONPath: Query Expressions for JSON, *Request for Comments*, RFC Editor, 2024. doi:10.17487/RFC9535. https://www.rfc-editor.org/info/rfc9535.

[36] P.-A. Champin, Sophia: A Linked Data and Semantic Web toolkit for Rust, Taipei, TW, 2020. https://www2020devtrack.github.io/site/schedule.

[37] S.M. Oo, T. Verbeken and B.D. Meester, RMLWeaver-JS: An algebraic mapping engine in the KGCW Challenge 2024, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024), Hersonissos, Greece, May 27, 2024*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3718, CEUR-WS.org, 2024. https://ceur-ws.org/Vol-3718/paper8.pdf.

[38] C. Stadler and S. Bin, KGCW2024 Challenge Report: RDFProcessingToolkit, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024), Hersonissos, Greece, May 27, 2024*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3718, CEUR-WS.org, 2024. https://ceur-ws.org/Vol-3718/paper13.pdf.

[39] M. Scrocca, A. Carenini, M. Grassi, M. Comerio and I. Celino, Not Everybody Speaks RDF: Knowledge Conversion between Different Data Representations, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024), Hersonissos, Greece, May 27, 2024*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3718, CEUR-WS.org, 2024. https://ceur-ws.org/Vol-3718/paper3.pdf.

[40] M. Freund, S. Schmid, R. Dorsch and A. Harth, FlexRML: A Flexible and Memory Efficient Knowledge Graph Materializer, in: *The Semantic Web*, A. Meroño Peñuela, A. Dimou, R. Troncy, O. Hartig, M. Acosta, M. Alam, H. Paulheim and P. Lisena, eds, Springer Nature Switzerland, Cham, 2024, pp. 40–56. ISBN 978-3-031-60635-9.

[41] M. Freund, S. Schmid, R. Dorsch and A. Harth, Performance Results of FlexRML in the KGCW Challenge 2024, in: *Proceedings of the 5th International Workshop on Knowledge Graph Construction co-located with 21th Extended Semantic Web Conference (ESWC 2024), Hersonissos, Greece, May 27, 2024*, D. Chaves-Fraga, A. Dimou, A. Iglesias-Molina, U. Serles and D.V. Assche, eds, CEUR Workshop Proceedings, Vol. 3718, CEUR-WS.org, 2024. https://ceur-ws.org/Vol-3718/paper9.pdf.

[42] D. Van Assche, D. Chaves-Fraga, A. Dimou, U. Serles and A. Iglesias, KGCW 2024 Challenge @ ESWC 2024, Zenodo, 2024. doi:10.5281/zenodo.11577087.

[43] E. Prud'hommeaux and G. Carothers, RDF 1.1 Turtle, W3C Recommendation, W3C, 2014, https://www.w3.org/TR/2014/REC-turtle-20140225/.

[44] R. Cyganiak, A relational algebra for SPARQL, *Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170* **35**(9) (2005).

[45] E. Prud'hommeaux and C.B. Aranda, SPARQL 1.1 Federated Query, W3C Recommendation, W3C, 2013, https://www.w3.org/TR/2013/REC-sparql11-federated-query-20130321/.

[46] D. Dell'Aglio, A. Polleres, N. Lopes and S. Bischof, Querying the Web of data with XSPARQL 1.1, *CEUR Workshop Proceedings* **1268** (2014), 113–118.

[47] D. Chaves-Fraga, F. Priyatna, A. Cimmino, J. Toledo, E. Ruckhaus and O. Corcho, GTFS-Madrid-Bench: A benchmark for virtual knowledge graph access in the transport domain, *Journal of Web Semantics* **65** (2020), 100596. doi:https://doi.org/10.1016/j.websem.2020.100596. https://www.sciencedirect.com/science/article/pii/S1570826820300354.

[48] M. Scrocca, M. Comerio, A. Carenini and I. Celino, Turning Transport Data to Comply with EU Standards While Enabling a Multimodal Transport Knowledge Graph, *The Semantic Web – ISWC 2020* (2020), 411–429. ISBN 9783030624668. doi:10.1007/978-3-030-62466-8_26.

[49] B. De Meester, T. Seymoens, A. Dimou and R. Verborgh, Implementation-independent function reuse, *Future Generation Computer Systems* **110** (2020), 946–959. doi:https://doi.org/10.1016/j.future.2019.10.006. https://www.sciencedirect.com/science/article/pii/S0167739X19303723.