Automated Configuration of Schema Matching Tools: A Reinforcement Learning Approach

Michał Patryk Miazga[1], Daniel Abitz[1, 2], Matthias Täschner[1], and Erhard Rahm[1,3]

Abstract: Schema matching involves identifying matching relationships between different data schemas. While this task is supported by semi-automatic tools, achieving optimal results requires configuring such tools which can be challenging depending on the number of configuration options and schema characteristics. This study proposes a novel approach utilizing Reinforcement Learning (RL) to automate the configuration of schema matching tools. RL has proven to be well-suited for complex optimization problems but has not yet been applied for schema matching. We outline how the configuration of a schema matching tool can be tackled as an RL task and how the corresponding learning process can be accelerated and optimized. We evaluate the RL approach for a large real-world dataset and show that it can be applied to different matching tools.

Keywords: Reinforcement Learning, Schema Matching, Action Masking, Automatic Configuration

1 Introduction

The task of matching semantically equivalent elements of data schemas is of high importance, especially for data transformations between different formats and for data integration. Integrating and fusing data from multiple sources with differing data types, structures, schemas, or ontologies is needed in numerous domains and application areas. Schema matching is therefore a vital step for data integration, along with other tasks such as data cleaning or entity resolution. It is also needed in the construction and refinement of knowledge graphs, e.g., when new data sources need to be integrated into an existing knowledge graph [Ho24].

The ever-increasing number and volume of data sources make manual matching impractical and time-consuming. Hence, there is a strong need for automated or semi-automated matching approaches and user-friendly matching tools [Ra11; Ra16].

As indicated in Figure 1, these matching tools typically support a multitude of schema match approaches (or matchers) and their combination. Matchers determine the similarity of pairs of elements (e.g., attributes) from the input schemas. The similarity values of multiple matches can be aggregated to derive combined similarities for a final selection of match candidates (that can be verified by a human expert). Individual matchers either operate on the schema level, instance data level, or a combination of both. Matching on the schema

[1] Center for Scalable Data Analytics and Artificial Intelligence (ScaDS.AI) Dresden/Leipzig, Universität Leipzig, Germany, michal.miazga@uni-leipzig.de
[2] University Computing Centre, Leipzig University, Germany
[3] Department of Computer Science, Leipzig University, Leipzig, Germany

level typically involves the analysis of linguistic and structural information while matching on the instance data level relies on the examination of the data values of schema elements. Matchers may also make use of background knowledge, embeddings, and machine learning, e.g., to determine instance-level match candidates [Ay22].

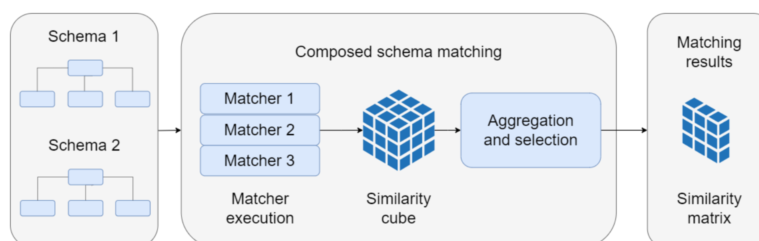


Fig. 1: A simplified view of a schema matching tool combining several matchers (inspired by COMA [DR02]).

Tools supporting such a composed schema matching, i.e., that combine multiple matchers, have shown to perform very well and they are also a necessity since no single matching technique is universally effective for all kinds of schemas and domains [Al17; Ra11; RB01]. However, the potential of such tools can only be utilized if they are configured appropriately, i.e. according to the characteristics of the schemas and instance data. This poses new challenges for users and demands extensive information on domain knowledge of the data and how the different matching approaches work. In particular, it has to be decided which matchers should be applied and whether to put them in parallel or in a certain order. Furthermore, each matcher as well as the aggregation and selection steps need to be configured leading to a particularly high number of possible configurations of different effectiveness.

The use of Reinforcement Learning (RL) for schema matching configuration is an area which has to our knowledge not yet been applied. In this paper, we explore whether RL can configure schema matching tools as well as, or better than, human experts. We therefore aim to automate the configuration of composed schema matching. In particular, we determine the relevant actions to consider for the configuration and as a suitable reward mechanism and approach to achieve an effective iterative optimization. We evaluate our new RL-based configuration approach for a real-world dataset. We also show how the presented RL approach can be applied to two matching tools, AgreementMaker Light [Fa13; Fa14; Fa19] and LogMap2 [JC11; Ji12].

The next section reviews related work on schema matching tools and the use of RL for auto-tuning. Section 3 provides a problem definition and background information about the chosen RL model. Section 4 outlines our RL approach for composed schema matching in detail. Section 5 contains the evaluation, including the application of the approach to different matching tools. We give a summary and suggestions for future research in Section 6.

2 Related Work and State of the Art

2.1 Schema Matching

Over the past 25 years, significant research efforts have been dedicated to schema matching and ontology alignment. Numerous approaches and tools have been developed in this domain, as described in several surveys [BMR11; OYD21; RB01]. The schema matching tool COMA (stands for **CO**mbined **MA**tching) has pioneered the multi-matcher approach more than two decades ago [DR02; Ma11]. This approach has then been followed by numerous further tools. Several recent approaches used machine learning to realize stand-alone matchers [Ay22; Fe18; KKK18; SGR20]. For example, LEAPME uses supervised machine learning to match schema properties based on instance characteristics and features of the property name represented by word embeddings.

Another recent development is the use of large language models (LLM) [HP23; Pa24; SS24] where schema matching is achieved by providing data along with carefully designed prompts to obtain the most accurate results. The LLM-based approaches are promising but still in their infancy, especially regarding necessary optimization techniques.

Several previous studies addressed already the optimization of schema matching configurations [BD11]. eTuner [Le07] uses a sequential, greedy approach called staged tuning to initially tune each matcher separately for a workload, and then optimizes the combination of the matchers. GATuner [FZY10] applies genetic algorithms to match against generated scenarios with known ground truth to find configurations that improve performance. Peukert et al. [PER12] use features extracted from input schemas and intermediate match results to define and adapt rules for selecting matchers and other operators. The Valentine framework [Ko21a; Ko21b] aims at finding good configurations of existing tools by naively evaluating a large number of possible settings for a few selected parameters but without dealing with more complex decisions such as the selection of matchers.

The discussion shows that the previous approaches to automatically configure schema matching tools are still limited. Some are computationally intensive, others are limited to configuring only a small subset of parameters, and no previous approach is capable of constructing complete match workflows (execution graphs) or scaling to a larger optimization space, as we will demonstrate in our approach using Reinforcement Learning (RL). We also show that our approach can be applied to different tools. We demonstrate this for two state-of-the-art multi-matcher tools, AgreementMaker Light (AML) [Fa13; Fa14; Fa19] and LogMap 2 [JC11; Ji12], that have been among the best ones in recent Ontology Alignment Evaluation Initiative (OAEI) match evaluations (e.g. in [Po21]). Furthermore, these tools can be configured so that RL is applicable (in principle).

AML is designed to align large-scale ontologies. It aims at scalability, extensibility, and satisfiability and can make use of external knowledge sources such as WordNet [Fe10]. AML leverages lexical matching algorithms as its foundation, completed by structural

algorithms for both matching and filtering, as well as its proprietary logical repair algorithm. While AML comes with a default configuration optimized for certain ontologies, manual configuration is possible allowing users to adjust parameters such as matching threshold, entity types to match, and instance matching mode [Fa13; Fa14; Fa19]. The ontology matching tool LogMap2 [JC11; Ji12] is particularly suited for large-scale and complex ontologies, such as in the biomedical domain. It leverages description logic reasoning like HermiT [MSH09] and Condor [SKH15] which are optimized for classification, afterwards computed classes are extended by additional information. It supports several approximate string matching techniques and structural ontology matching considering the hierarchy of classes and the relationships between entities. LogMap2 comes with a default configuration file that can be adjusted.

2.2 Reinforcement learning for configuration

In recent years, there have been notable advancements in using RL to configure or fine-tune systems in various domains. Consequently, it is worthwhile to examine the current approaches and applications of RL in this regard.

Some attention has been directed towards optimizing configuration parameters for complex systems, including auto-tuning mechanisms for blockchain systems [Li23] and distributed systems such as Apache Spark [HZZ22; VC18]. Wang et al. [Li23] introduces Athena, a performance auto-tuning system for Hyperledger Fabric [An18]. The parameter tuning challenge in Fabric is a multi-agent coordination problem that is effectively addressed by a so-called Permissioned Blockchain Multi-Agent Deep Deterministic Policy Gradient (PD-MADDPG) algorithm. Huang et al. [HZZ22] utilize a deep-learning neural network and a modified Q-learning algorithm for tuning Apache Spark configuration parameters. Another study focuses on optimizing the configuration for Spark Streaming [VC18]. They utilize a large amount of training data and the selection of suitable metrics and automatic tuning using RL. The results show a significant reduction in latency compared to human-configured clusters.

RL has also been applied for auto-tuning of database systems [Ba16; Li19; Zh19]. Q-learning and Deep Q Network (DQN) algorithms have been explored in the domain of database optimization, demonstrating their effectiveness in system tuning, though limited by their discrete action space. Therefore, the Deep Deterministic Policy Gradient (DDPG) has been proposed to overcome the limitations of the action space of previous algorithms, allowing the use of continuous action spaces [Zh19].

Unlike prior applications of RL for configuration, the approach proposed in this paper diverges towards identifying the optimal arrangement of components within a configuration for composed schema matching. This entails determining both the combination of components and the adjustment of their corresponding parameters to enhance the effectiveness of the schema matching process.

3 Terms and definitions

3.1 Schema Matching

A schema is a formal description of the structure of data and defines the organization of data as a blueprint, including the definitions of elements, such as attributes of tables in relational databases, elements and attributes of an XML schema or DTD, or concepts in an ontology or knowledge graph. Instance data refers to the actual data or entities that conform to the structure defined by a schema. For example, the entities of a relational database R are tuples $(a_1, a_2, ..., a_n)$, where each a_i is an attribute value corresponding to attributes from the set of schema elements of R.

Schema matching is the task of determining semantic correspondences or matches between elements of two schemas S_1 and S_2 with sets of schema elements E_1 and E_2. A match indicates that two schema elements $e_1 \in E_1$ and $e_2 \in E_2$ are semantically equivalent and can be denoted as $m(e_1, e_2)$. Similarity is a quantitative measure of how high the equivalence of two schema elements is, where similarity of e_1 and e_2 is a function $sim(e_1, e_2) \rightarrow [0, 1]$. Thus, similarity serves as a quantitative basis for determining matches between pairs of elements.

To evaluate schema matching algorithms, a ground truth is crucial. Given two schemas S_1 and S_2, the ground truth is a predefined set of matches defined as $G = \{g(e_1, e_2) \mid e_1 \in E_1, e_2 \in E_2\}$ where $g(e_1, e_2)$ is a known (or presumed) match.

To evaluate our solution, we employ three primary evaluation metrics: the F1 score, the area under the receiver operating characteristic curve (ROC-AUC), and the Matthews Correlation Coefficient (MCC). These metrics are widely used to measure performance in binary classification tasks [Be19]. The F1 score is particularly valuable for imbalanced scenarios with an uneven distribution of classes, such as the sets of matching and non-matching pairs of schema elements in schema matching. ROC-AUC provides insight into the model's ability to distinguish between classes at different threshold settings and is commonly used to evaluate model performance [Be19; Br97; CJ07]. The MCC score offers a reliable assessment across all categories of the confusion matrix and effectively accounts for the proportions of both positive and negative classes [Be19; CJ20; CJ23].

3.2 Reinforcement Learning

Reinforcement Learning (RL) is one of the primary pillars of machine learning, standing alongside supervised and unsupervised learning. In contrast to supervised learning's reliance on labelled data, and unsupervised learning, where the model tries to find patterns without guidance, RL thrives on the principle of trial and error. For this purpose, an RL agent interacts with the learning environment to optimize it iteratively. The interaction of the agent and the learning environment is influenced by the following components:

- **Action Space**: This space is defined once and cannot be modified. All possible actions are defined at the beginning and outline the moves or actions an agent can take at any stage. The agent's actions can be discrete (such as go right, go left) or continuous (the agent can select an action from a range of values for each state, e.g., turn the steering wheel a few degrees).
- **Observation Space**: This refers to the information or state that the agent can perceive about its environment while taking actions. Through this, the agent learns how its actions impact the environment.
- **Reward**: The environment returns the reward in the form of a scalar value it informs the agent about its progress in the environment after each movement.

In RL, agents engage within the environment, receiving rewards or penalties as feedback based on the outcomes of their actions. The agent aims to amass cumulative rewards over time, linked to a policy set of guidelines dictating action selection in various scenarios. A policy dictates how an agent should behave by specifying the actions it should take in each state. The rewards serve as guiding beacons, pointing to advantageous actions. Through a blend of exploration and exploitation, the agent accumulates knowledge, refining its decision-making efficiency over time. For this, there are a variety of algorithms. One of these is Proximal Policy Optimisation (PPO), an RL algorithm within the gradient-based methods category, designed to iteratively refine the policy. It updates the policy by maximizing a surrogate objective function while constraining changes to prevent excessive deviation from the previous policy, thus balancing exploration and exploitation. By using surrogate losses, PPO aims to improve the agent's action selection strategy and address stability issues [Sc17]. Compared to previous applications of RL for configuration, the Deep Q-Networks (DQN) algorithm is not suitable for action spaces that modify multiple parameters simultaneously due to its inability to handle complex, multi-dimensional action spaces efficiently. Implementing DQN for each parameter introduces significant complexity and computational overhead, as it requires managing separate Q-value tables or network outputs for each action combination. Compared to DQN, recent research indicates that PPO performs better at solving complex tasks [DG24].

4 Our approach

For the development of the RL approach, we implemented a configurable schema matching tool that we describe first before we outline the RL approach.

4.1 The Configurable Schema Matching Tool

The architecture of the **S**chema **M**atching **T**ool (SMT) is shown in Figure 2. The input is expected to be a pair of schemas and a configuration for the matching process, both in a

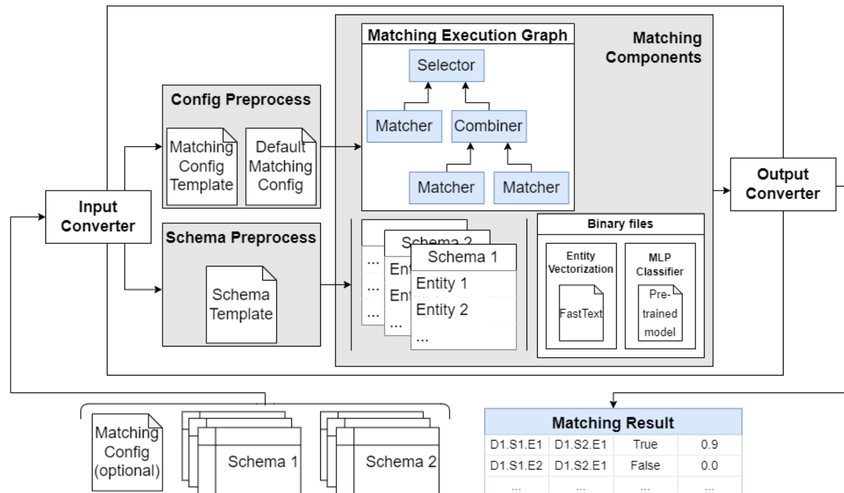


Fig. 2: Coarse architecture of our schema matching tool.

predefined structured format. The matching is executed by components which compute and aggregate similarities for pairs of schema elements. The final matching results, determined by a selector component, are returned as output. So, for each input pair of schema elements (e_1, e_2), an output is generated as tuple (e_1, e_2, m, sim), where m is a Boolean value determining whether or not this pair is a match, and sim is the computed aggregated similarity value for this pair also describing the confidence of the matchers in this regard.

Our tool incorporates several matching components that can be selected and configured by the RL agent, including:

- **Matching Approaches or Matchers**, in particular, name matching based on string similarity of element names, instance data matching based on aggregated instance data properties, clustering-based matching (K-Means and Mean-Shift variations), and match prediction with pre-trained Multi-Layer-Perceptron (MLP) classifiers (inspired by [Ay22]).
- **Aggregation Methods**, determining the average, maximum or minimum of similarity values, or distance weighted aggregation (weighted average based on distance metrics).
- **Selectors** deciding whether a pair of schema elements is a match based on the computed similarity values for this pair, with either threshold-based selection (applies a threshold either to each computed similarity value or, optionally, to the average of these values) or variance-based selection (applies a threshold both to the computed similarity values and to the variance over all these values).

The matching configuration specifies the matching process which is represented as a tree-like

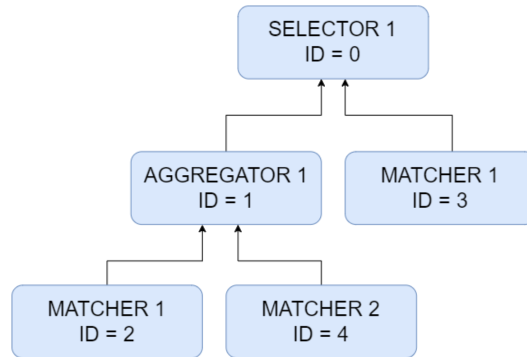


Fig. 3: Tree-like structure of a composed matching configuration.

execution graph of the selected components, as shown in an example shape in Figure 3, where the following rules apply:

- R1: A matcher is of type leaf (no children are allowed),
- R2: An aggregator is of type (inner) node and must have $n \geq 1$ children which must be of type node or leaf,
- R3: A selector is of type root and must have $n \geq 1$ children which must be of type node or leaf, and
- R4: There is exactly one root.

The matching process is executed in a topological order starting from the leaves and progressing to the root with computed or aggregated similarities passed to the parent component.

4.2 Reinforcement Learning Approach for Configuration

In this section, we explain our RL approach which is embedded into the architecture shown in Figure 4. As illustrated, the **PPO Agent** engages with the **Environment** by executing actions. For each **Action** (see sections 4.2.1 and 4.2.2) performed, the agent receives a **Reward** (section 4.2.4) and a current progress **Observation** (section 4.2.3). Once the agent accumulates a sufficient number of actions that constitute a valid execution tree, these actions are forwarded to the **Result Processor**. The **Result Processor** is responsible for converting the actions into an execution tree. Subsequently, it uses the configuration to execute the **SMT** and obtain the match results for this configuration. These results are then compared against the **Ground Truth** and reported back to the **Environment** as performance in the form of various **Measures** (section 3.1).

In Section 4.2.5, we outline additional termination constraints while Section 4.2.6 describes the approach to balance between known strategies (exploitation) and the discovery of new options (exploration).

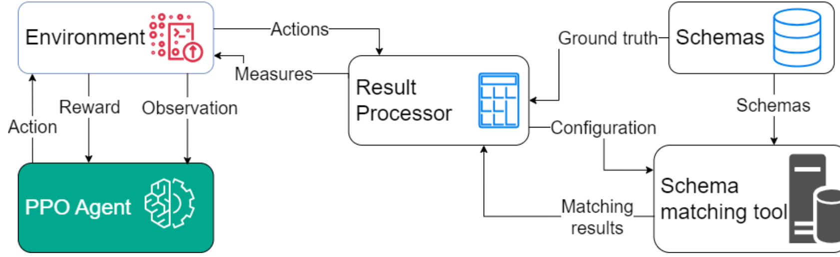


Fig. 4: Architecture overview for our RL environment for configuration.

4.2.1 Action Space Partitioning

One of the key challenges in setting up the RL environment is to define and designate the configuration components constituting the action space for the agent. This includes the selection of matching components of different types (matcher, aggregator, selector), their parameters and execution order. For our PPO agent, these multi-dimensional action spaces must be flattened into a one-dimensional sequence which is also needed to apply the optimization of dynamic action masking to be described in section 4.2.2.

To define the action space, let $c_t \in \{0\} \cup \{1, ..., |c_t|\}$ be a component of type $t \in T$, T the set of available types and $|c_t|$ the number of available components of type t. The value 0 is a special case indicating that no component of type t has been selected. Further, let p_{t,c_t} be the parameter array of c_t and $|p_{t,c_t}|$ the corresponding array length. The array $P_t = p_{t,1} + ... + p_{t,|c_t|}$ is the concatenation of all parameter arrays of type t and $|P_t| = \sum_{i=1}^{|c_t|} |p_{t,i}|$ the length of the concatenated array. To connect components in a tree-like hierarchy as shown in Figure 3, an additional parameter $p_id \in \{0, ..., max_comp\}$ is introduced to specify the parent component (action) in the execution tree; max_comp refers the maximum number of possible components in an execution tree. To link actions, they are assigned a unique ID (index position) based on the order in which they are created. The first action is assigned ID 0, the second action ID 1 and so on. For the first action, p_id is set to 0 meaning that it points to itself. Since each configuration has exactly one root, this special case can be ignored when building the final configuration. It is important to note that an action represents the decision as to which a specific component was chosen. Consequently, an action is only valid if exactly one c_t is not equal to 0.

The SMT presented in Section 4.1 consists of three different component types $T = \{s, a, m\}$ (selector s, aggregator a, and matcher m). Following the general approach to action space construction, the corresponding action space can be created as shown in Table 1.

Position	Value	Description
0	c_s	Chosen selector s.
$1 \cdots \lvert P_s \rvert$	P_s	Parameters for selector (optional).
$1 + \lvert P_s \rvert$	c_a	Chosen aggregators a.
$2 + \lvert P_s \rvert \cdots 1 + \lvert P_s \rvert + \lvert P_a \rvert$	P_a	Parameters for aggregators (optional).
$2 + \lvert P_s \rvert + \lvert P_a \rvert$	c_m	Chosen matchers m.
$3 + \lvert P_s \rvert + \lvert P_a \rvert \cdots 2 + \lvert P_s \rvert + \lvert P_a \rvert + \lvert P_m \rvert$	P_m	Parameters for matchers.
$3 + \lvert P_s \rvert + \lvert P_a \rvert + \lvert P_m \rvert$	p_id	id of parent component.

Tab. 1: Action space for the three component types selector s, aggregator a, and matcher m with their corresponding parameter array P, array length $|P|$, and the connecting value p_id represents the maximum number of components in the execution tree.

To illustrate the action space according to these principles, we present the steps the agent takes to create the configuration shown in Figure 3. These steps are outlined in Listing 1. In this simplified configuration, we include a parameter for the selector and the second matcher assuming default parameters for the other components. The action space, in the beginning, has six parts (index positions 0 to 5) to specify the possible values for the three component types as well as for the maximal number of components to be created. For the initial selector specification (id 0) the possible values are 0 (no selector) or 1 assuming only one possible kind of selector. For the selector parameters (id 1) there is either no parameter (0) or possible similarity threshold values between 1 and 101. Index position 2 defines the space for adding an aggregator, with 0 for none followed by either 1 or 2 for two possible kinds of aggregators. Index 3 defines the space for adding a matcher, using the same values: 0 for none, 1 for matcher 1, and 2 for matcher 2. Index 4 defines the space for parameters related to matcher 2, while index 5 defines the possible values for linking to the parent up to the maximum number of components in the execution tree (10, in the example). As can be seen in the lower part of Listing 1 there are two instances of matcher 1 with either parent 1 (line 10) or 0 (line 12).

```
1  Defined Action Space:
2  [[0,1],[0,1,...,101],[0,1,2],[0,1,2],[0,1,...,5],[0,1,...,10]]
3
4  The agent steps are interpreted as follows:
5  # Adds a Selector 1 id = 0 with parameter 80.
6  Step 1: [1,80,0,0,0,0]
7  # Adds an Aggregator 1 id = 1, linked to the parent with p_id = 0.
8  Step 2: [0,0,1,0,0,0]
9  # Adds a Matcher 1 id = 2, linked to the parent with p_id = 1.
10 Step 3: [0,0,0,1,0,1]
11 # Adds a Matcher 1 id = 3, linked to the parent with p_id = 0.
12 Step 4: [0,0,0,1,0,0]
13 # Adds a Matcher 2 id = 4 with parameter 1,
```

```
14    # linked to the parent with p_id = 1.
15    Step 5: [0,0,0,2,1,1]
```

List. 1: Simplified action space and agent steps within that space for configuration of Figure 3.

4.2.2 Dynamic Action Masking

To accelerate the learning process, we employ action masking [HO20; Ta20] to dynamically adjust the available actions based on the agent's preceding moves and current state. This strategy aims to prevent the agent from attempting redundant actions, such as adding a selector despite one having already been chosen in a prior step.

Algorithm 1 Action Masking

1: Initialize $action_space_map$ for the flattened action space.
2: $i \leftarrow 0$
3: **for** each action a in the action space **do**
4: $\quad action_space_map[a] \leftarrow [i, i + length(a)]$
5: $\quad i \leftarrow i + length(a)$
6: **end for**
7: Initialize $action_mask$ array of size i with all elements set to True.
8: **for** each action a in $action_space_map$ **do**
9:     **if** a is neither a selector nor a parameter of a selector **then**
10:         **for** each idx in $action_space_map[a]$ **do**
11:             $action_mask[idx] \leftarrow (idx == 0)$
12:         **end for**
13:     **else**
14:         **for** each idx in $action_space_map[a]$ **do**
15:             $action_mask[idx] \leftarrow (idx \neq 0)$
16:         **end for**
17:     **end if**
18: **end for**
19: Add $action_mask$ to the observation.

To implement the masking mechanism the action space is treated as a set of probabilities, where each component's likelihood of selection is dynamically adjusted. For instance, if a selector had already been chosen in a previous step, the likelihood of adding another selector in subsequent steps is reduced to zero. Similarly, if an aggregator was introduced, the parent ID list is modified to accommodate a newly feasible action, with its likelihood adjusted to match the available actions. To realize this dynamic adjustment, a method was devised to update the probabilities of each action after every step. Algorithm 1 illustrates the initialization of action masking within a multi-discrete action space. This masking occurs exclusively during the initial step of selector selection. This involves mapping the probabilities to their corresponding actions, modifying them accordingly, and subsequently incorporating them into the agent's decision-making process. In subsequent steps, the roles are inverted, and the selectors, along with their parameters, undergo masking.

The masked space then corrects the action probabilities, which are subsequently updated by the forward progression function by [HO20; Ta20]. This ensures that only feasible and relevant actions are considered at each stage of the learning process.

4.2.3 Observation Space

The observation space comprises two distinct parts: the first part consists of an array holding all possible actions, encoded with 0 or 1 for masking (see Section 4.2.2), while the second part provides essential information for the agent. In our scenario, the second part of the observation space is an initially empty area that records the current state of the execution graph. With each step the agent takes, this space is updated to include the component that the agent introduced in the preceding step.

Algorithm 2 Initialize observation space (obs) with action masking

1: $obs \leftarrow \{\}$
2: $obs['action_masking'] \leftarrow action_mask$
3: $obs['observations'] \leftarrow \{\}$
4: **for** i **from** 0 **to** $max_components$ **do**
5: $\quad obs['observations'][i] \leftarrow \{\}$
6: $\quad obs['observations'][i]['component'] \leftarrow 0$
7: $\quad obs['observations'][i]['parent'] \leftarrow 0$
8: $\quad obs['observations'][i]['parameters'] \leftarrow [0, \ldots, 0]$
9: **end for**

Algorithm 2 presents the initialization of the observation space with action masking. The observation space contains all valid steps where i is the step number at which the component was added. Each component within this construct embodies its distinct parameters. To save space, only the corresponding parameter array p_t will be stored and not the whole array P_t. For the observation space to still have a constant size, the initial array of 'parameters' is set to $max_t(\sum_{j=1}^{|t|} |p_{t,j}|)$ (the largest sum of the sizes of the parameter arrays t). After the initialization process, the final values are set according to the following steps:

1. $'component' = t \in T \rightarrow \mathbb{N}$
2. $'parent' = p_id$
3. $'parameters' = p_{t,e_t}$ with size $max_t(\sum_{j=1}^{|t|} |p_{t,j}|)$

Step (1) maps the component type t to a natural number > 0. The linking value n is set as parent according to step (2). Finally, the parameter array p_t is set in step (3). It is important to ensure that the length of p_t is equal to the maximum sum of the sizes of arrays of parameters of any component t to accommodate any parameter. The observation space for the execution graph shown in Figure 3 is represented by up to max_comp components shown in Listing 2.

```
{
  0: {'component': 1, 'parent': 0, 'parameters': ...},
  1: {'component': 3, 'parent': 0, 'parameters': ...},
  2: {'component': 5, 'parent': 1, 'parameters': ...},
  3: {'component': 5, 'parent': 0, 'parameters': ...},
  4: {'component': 6, 'parent': 1, 'parameters': ...},
  5: {'component': 0, 'parent': 0, 'parameters': ...},
  ...
  max_comp: {'component': 0, 'parent': 0, 'parameters': ...}
}
```

List. 2: Observation space for the example configuration in Figure 3, combined with the action space definition provided in Listing 1. In this configuration, component '1' corresponds to Selector 1, component '3' to Aggregator 1, component '5' to Matcher 1, component '6' to Matcher 2, and component '0' represents a component not yet defined.

4.2.4 Reward system

In the field of agent-based systems, decision-making and the concept of reward are crucial. After executing a validated action, an agent receives a positive reward for feasible actions and incurs a penalty for disallowed ones at the given moment. For instance, if an agent attempts to add multiple components to the configuration at once such as trying to add both a matcher and an aggregator simultaneously, it will receive a penalty.

The first step of the agent involves adding a selector to the configuration tree, which serves as the starting point for subsequent actions. Additionally, the selector-specific parameters must be defined, and the agent is rewarded for successfully adding and defining a selector. In the following step, the agent has two main options to choose from: adding a matcher or an aggregator and defining their respective parameters. In both cases, the agent must not only make this selection but also determine the identifier of the parent. In the second step, this only relates to the initially added selector. In subsequent steps, the list of available parents expands to include previously added aggregators.

As the process continues, subsequent steps replicate the available actions following the choice of the selector. Once the necessary minimum criteria, as outlined in the configuration rules, have been met, all steps undergo validation to confirm compliance with the configuration file's specifications. For instance, putting a selector and at least one child, or, in the case of a selector and aggregator, ensuring the aggregator also has at least one child. After successful validation, the resulting configuration file is transmitted to the SMT alongside the schemas to execute the matching process. The matching result for this configuration file is then compared with the ground truth available for those schemas (see Section 5.1). The ground truth does not need to involve hundreds of tables, a small sample, such as two matched

schemas, is sufficient. The agent learns primarily the characteristics of the matching tool for a particular type of schema, not the schemas themselves.

To optimize and balance the various metrics, we use the **G**eometric **M**ean of the F1 score (calculated for binary values (0 and 1) after applying a threshold by the agent) and ROC-AUC values (which evaluates the similarity of values between 0 and 1) as a metric for aggregated **P**erformance (GMP score), see Formula 1. By merging these metrics into a single GMP score, the agent can optimize its performance based on both predictive performance and the impact of different component configurations on similarity scores. Ultimately, the reward is added to the total collected by the agent for assembling a valid configuration.

$$\text{GMP score} = \sqrt{\text{F}_1 \times \text{ROC-AUC}} \tag{1}$$

4.2.5 Termination constraints

To reduce the likelihood of the agent reaching a local optimum, where the balance between reward and penalty is finely tuned, two constraints were established:

- Firstly, a limit was imposed on the **M**aximum **N**umber of components (MN) allowed in the configuration, defined upon the environment's creation. This restriction ensures that the agent cannot exceed the predetermined **N**umber of **C**omponents (NC), although it does not impose further limitations on the configuration tree's structure. When reaching the specified number of components in the configuration, the event is terminated, and a signal is transmitted along with the response to the corresponding event step.
- Secondly, a constraint called a number of **A**vailable **L**ives (AL) mitigates the risk of the agent getting trapped in a local optimum. This constraint reduces the available lives for each incorrect move or repetitive execution of the same incorrect action. When all available lives are exhausted, the trial terminates, and a penalty is incurred.

The termination condition can be described as shown in Formula 2. If the episode is terminated, it serves as a strong signal for the agent to seek other solutions. It has been observed that an optimal termination strategy involves setting conditions that support the exploration of diverse moves by the agent while mitigating the risk of stagnation.

$$\text{Terminate Trial} = \begin{cases} \text{True}, & \text{if } (\text{NC} > \text{MN}) \lor (\text{AL} \leq 0) \\ \text{False}, & \text{otherwise} \end{cases} \tag{2}$$

4.2.6 Exploitation vs Exploration

The PPO algorithm was selected for training the agent thanks to several key factors. Firstly, PPO's implementation of proximal optimization effectively limits policy changes during updates minimizing the risk of large fluctuations and enhancing stability — a crucial requirement in RL [Sc17]. In PPO, entropy measures the randomness of the policy's action distribution, encouraging exploration by preventing the policy from becoming too deterministic. To maintain a balance between exploration and exploitation, entropy regularization is employed to prevent the agent from converging to a local optimum [Ah19]. This approach ensures the agent explores various components and their combinations in the action space.

Configuring the PPO algorithm correctly was imperative given the unique characteristics of our environment. For our tool to yield consistent results for the same configuration and input data, a delicate balance between exploitation and exploration is required. While it is important to explore various combinations of components in the configuration tree, leveraging known components which yield favourable results is equally essential. This balanced approach maximizes the efficiency of the learning process while facilitating comprehensive exploration of the action space. Given the unique characteristics of our environment, the aim is to prioritize exploration over exploiting known actions. Increasing the entropy coefficient encourages the agent to explore more actions but a balance has to be found to avoid excessive randomness. Entropy measures the uncertainty or information in potential actions and outcomes. To address this, an entropy bonus, as discussed in [Mn16; Sc17; SCA17], is applied to encourage more diverse action selection.

5 Evaluation

To assess whether our RL-based approach for the configuration of schema matching tools produces results comparable to or better than those of expert configurations, we begin by providing an overview of the dataset and the evaluation setup. We then present the training process, and the results achieved using our RL approach on this dataset. Finally, we investigate the application of our approach to other configurable tools like AML and LogMap2.

5.1 Dataset and evaluation setup

To evaluate our **S**chema **M**atching **T**ool (SMT) on real-world schemas, we utilize the Web Data Common Schema Matching Benchmark [We23]. We test the SMT by transforming the schemas to the predefined format, including a column header and aggregated schema instance statistics. The dataset comes in two variants: *T2D-SM-WH* and *T2D-SM-NH*. The T2D-SM-WH dataset provides both instance data and descriptive column headers. In

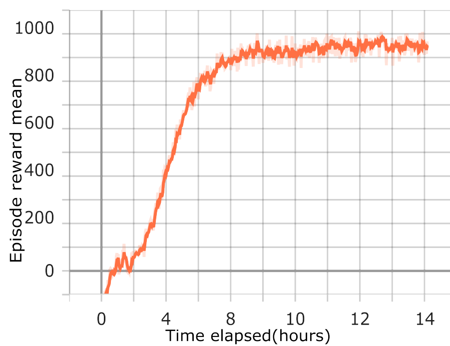
contrast, the T2D-SM-NH dataset contains instance data without meaningful or descriptive header names. Each of the two datasets contains 356 tables from 28 thematic domains which are divided into three parts: the test set includes two subsets with 89 tables each, the train set comprises two subsets with 71 tables each, and the validation set contains two subsets with 18 tables each. When evaluating the test set from the WDC Schema Matching Benchmark, the SMT needs to compare up to 212,004 schema elements highlighting the significant matching complexity.

For our setup, we utilized a machine equipped with 64 GB of RAM and an AMD Ryzen 9 7950X 16-core processor. The Reinforcement Learning environment was developed in Python, leveraging the RLlib [Li18] library, along with Gymnasium [To23] and scikit-learn [Pe11] for basic evaluation metrics.

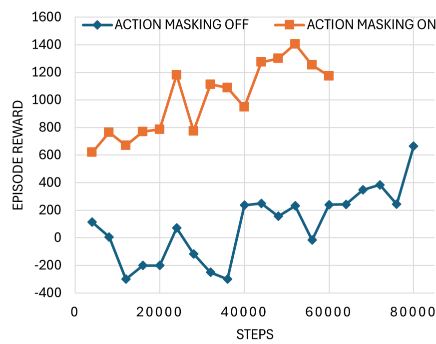
5.2 Training

The training of the agent extends through two distinct phases, each aimed at addressing specific aspects of its learning process.

In the initial phase, the primary objective is to familiarize the agent with the difficult process of assembling a valid execution graph. This involves imparting knowledge on the number of components to include and the complexity of the execution graph. To encourage the agent to learn, a reward structure was introduced, where rewards are granted based on the number of components successfully added to the configuration file and the advancement of the resulting execution graph. However, it was crucial to strictly adhere to the requirements for the validity of the configuration at this phase. The underlying goal is to equip the agent with the necessary knowledge to navigate the environment effectively including tasks such as selecting and adding components, referencing parents, or ensuring the uniqueness of components within each level of the execution graph.



(a) Average agent rewards during training.



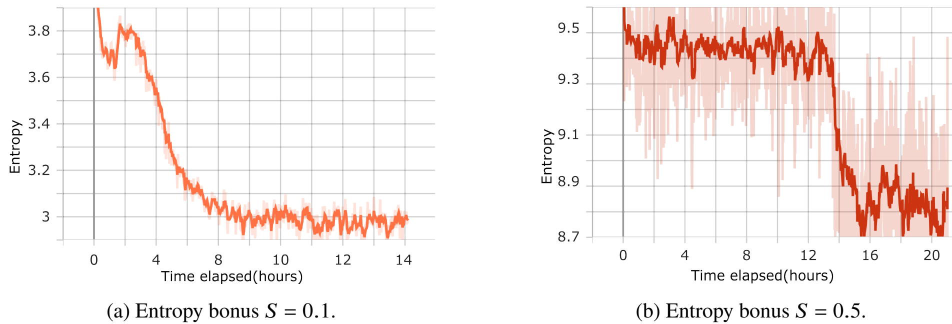
(b) Agent performance with action masking.

Fig. 5: Agent Training Results.

In the second training phase, the focus shifts from quantity to quality. The reward for adding components is reduced, and the agent's reward is extended on how well the configuration achieves the desired matching results. This phase refines the agent's decision-making to optimize matching outcomes. We ran the training for 14 hours. The average episode reward value fluctuated until the end of the initial 2-hour training period (with mean reward below 100) coinciding with the agent's transition into the second training phase, as shown in Figure 5a. Following the transition into the second phase, the average reward steadily increased until approximately 8 hours into the training process. In contrast to traditional agent training, which focuses on teaching an agent to perform specific actions, our approach aims to identify the optimal configuration. Thus, we maintain a scoreboard that records only the highest-performing configurations of the schema matching process.

Thanks to action masking, described in Section 4.2.2, the agent makes correct decisions when constructing the execution graph, leading to a high initial reward. For our test case, the agent requires approximately 80,000 steps without action masking to achieve a level of rewards comparable to an agent with action masking enabled from the start, see Figure 5b.

The entropy should gradually decrease during successful training, as demonstrated by applying an entropy bonus of $S = 0.1$, as shown in Figure 6a. As described in Section 4.2.6, in this context, we define S as the entropy bonus coefficient applied to encourage exploration. In contrast, with $S = 0.5$, decisions remain highly random, indicating limited progress despite extended training time, as illustrated in Figure 6b.



(a) Entropy bonus $S = 0.1$.

(b) Entropy bonus $S = 0.5$.

Fig. 6: Comparison of entropy with different entropy bonus S values.

The fine-tuning of RL agent configurations is not all necessary to find a good configuration for the schema matching tool. The experiments are solely intended to reduce the time required for the agent to identify a suitable configuration.

5.3 Results for SMT

A manual approach to finding an optimal configuration for our SMT matching tool based on trial and error would be extremely time-consuming and repetitive. Alternatively, another

solution is using a script to automate the generation of all feasible configurations. However, this method necessitated the imposition of certain constraints such as limiting the number of components to be examined and predefining the structure of the execution graph. Still, the number of generated configurations escalated quickly even with constraints applied. Given a simple graph structure with depth 1, the number of different combinations is $\#combinations = ((\#matcher_types)^{\#children})^{\#selector_types}$. Considering that multiple parameters can be set for each type of selector and matcher; the complexity further grows. This complexity expands when aggregators introduce another layer to the graph. Consequently, the total number of combinations quickly escalates to several million, even before considering nested aggregators. Moreover, due to the execution time required for each configuration, finding the optimal combination remains challenging, even with parallel execution. With a rough estimate of 20 trials per second, the execution time would take many weeks. Therefore, one of the evaluation points is the execution time for finding an optimal configuration.

Dataset	Time Action Masking Off	Time Action Masking On
T2D-SM-WH	1d 10h	6h
T2D-SM-NH	1d 2h	4h

Tab. 2: Time Comparison with and without Action Masking.

	T2D-SM-WH		T2D-SM-NH	
	Manual Configuration	RL Configuration	Manual Configuration	RL Configuration
F1 score	0.496	0.686	0.262	0.565
ROC-AUC	0.742	0.837	0.637	0.821
MCC	0.509	0.621	0.166	0.451
Accuracy	0.884	0.894	0.755	0.845
GMP score	0.607	0.758	0.409	0.681

Tab. 3: Matching quality results for manual and RL-optimized configurations.

The proposed RL approach avoids these excessive and impractical execution times, especially when action masking is applied. Table 2 shows the execution times to find well-performing configurations for the two datasets without and with action masking. Action masking leads to a drastic reduction of execution times by 22-28 hours to only 4-6 hours. The matching effectiveness results in Table 3 are based on the test data, using the best configuration identified by our method after training on the training portion of the T2D-SM-WH and T2D-SM-NH datasets. Table 3 indicates that the configuration produced by RL surpasses the manually adjusted configuration in terms of matching quality. This method proved particularly effective for T2D-SM-NH, where the absence of column headers in the dataset made the manual configuration of our tool challenging, making it difficult to obtain satisfactory results. However, the configuration determined through RL led to notable improvements, increasing the GMP score. The configurations generated by the agent were subsequently compared against those that had been manually selected and fine-tuned for each dataset.

5.4 Application to other Tools

To demonstrate the general applicability of the proposed RL approach, we apply it to other configurable tools that have been successful in the OAEI ontology matching contests [On23a]. All evaluations are conducted using the 2023 OAEI anatomy track dataset [On23b]. A direct comparison of our tool's results with the other ontology matching tools is not feasible as the OAEI anatomy dataset does not include the instance data required by our tool. Its matcher components primarily use aggregated information about the instance data for the similarity computation while structure-based matchers are not implemented. Following an initial assessment of available libraries, *AML* (AgreementMakerLight) in version 3.2 and *LogMap2* were chosen for further evaluation. Certain conditions must be met when exploring potential applications of our approach, including the ability to configure the execution of the matching processes, which requires developers to provide configuration options rather than coding them permanently into the code. In both cases, the RL environments for the tools were designed with a simplified implementation, as neither tool permits modification of the execution graph, only adjustments to their parameters. The configuration generated using RL was used in the manual mode of the AML tool and compared against the default configuration settings in the auto mode.

Tool	Configuration	Precision	Recall	F1 score	Time
LogMap2	Default Matching	0.911	0.847	0.878	-
	RL Configuration	0.934	0.844	0.887	1h
AML v3.2	Automatic Matching	0.957	0.881	0.918	-
	Default Matching	0.767	0.863	0.812	-
	RL Configuration	0.96	0.883	0.92	3h

Tab. 4: Performance metrics for different configuration approaches using LogMap2 and Agreement-MakerLight on the OAEI 2023 dataset.

It took 1 hour (LogMap2) and 3 hours (AML) for the RL model to produce configurations on par with the performance of the automatic mode, or in some cases even slightly better as shown in Table 4, which includes metrics supported by the evaluated tools. The learning process involves the agent selecting actions from the multi-discrete space, which are then converted into configurations and sent to AML for evaluation. Results from the evaluation mode are returned to the agent as a reward. The manual mode is predefined in the configuration, which contains universal settings for ontology matching. In contrast, the automatic configuration is a predefined mode that is permanently hard coded into the AML tool, fine-tuned for the OAEI. The automatic configuration facilitates the automatic selection of background knowledge sources [Fa14]. The manual mode serves as an optional configuration provided within AML for user customization and experimentation. Similar to the AML, we implemented the environment for the LogMap2 tool. The Automatic mode is the default mode for LogMap 2 [Ji12], it is defined in the configuration. Unfortunately, there is no proper documentation about the parameters inside the configuration, so only parameters for which it is possible to set a number, or Boolean values are available for modification for the agent.

6 Conclusion

We introduced a novel approach using Reinforcement Learning (RL) to automatically configure tools for composed schema matching. This approach presents a significant advance in the field offering a dynamic solution to improve the efficiency and quality of configuration. Previously, configuring a schema matching tool required in-depth domain knowledge and manual intervention which is time-consuming and complex. Wrapping the schema matching tool in the RL environment enables the automation of this process and allows the agent to adjust and optimize the configuration based on the available tool settings and data characteristics. One key aspect of our approach is the utilization of dynamic action masking within a multi-discrete action space, which allows the RL agent to focus on relevant actions and speeds up the learning process. Moreover, we proposed a reward mechanism based on the geometric mean of the metrics ROC-AUC and F1 score (GMP score) based on ground truth. By incorporating these metrics into the reward function, we aimed to optimize the matching outcomes. We have shown that a configuration generated by RL is not only faster to produce a good configuration compared to generating all possible combinations but has also proven to be better than a manually adjusted configuration. Additionally, a simplified version of RL-based automatic configuration was implemented and applied to other tools in the field, specifically AML and LogMap2 to demonstrate the general applicability of our approach. This involved designing an RL agent that learns to optimize the configuration parameters based on our proposed reward mechanism. The results of our experiments show that with the configuration generated by RL, the results obtained were equal to the default configuration and, in some cases, even better than the tool's default manual configuration or automatic matching mode, in terms of fitting accuracy. Allowing tools to be configurable enables the use of RL for automating configuration and constructing execution graphs. This approach could further extend to other tools requiring human fine-tuning.

In future work, we aim to compare the effectiveness of various RL algorithms in configuring schema matching tools. We want to assess their efficiency and speed in adapting configurations within schema matching tools through experimentation with different RL algorithms, such as Advantage Actor-Critic (A2C), Recurrent PPO, or Trust Region Policy Optimization (TRPO). Another possible implementation of our solution would be to integrate a pre-trained RL model into the running SMT. The pre-trained model would be capable of recognizing different input schemas and automatically selecting or adapting the optimal configuration for each specific type of schema.

Acknowledgement

The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany and by Sächsische Staatsministerium für Wissenschaft, Kultur und Tourismus in the programme Center of Excellence for AI-research „Center for Scalable Data Analytics and Artificial Intelligence Dresden/Leipzig“, project identification number: ScaDS.AI

References

[Ah19] Ahmed, Z.; Le Roux, N.; Norouzi, M.; Schuurmans, D.: Understanding the impact of entropy on policy optimization. In: International conference on machine learning. PMLR, pp. 151–160, 2019.

[Al17] Alwan, A. A.; Nordin, A.; Alzeber, M.; Abualkishik, A. Z.: A survey of schema matching research using database schemas and instances. Int. Journal of Advanced Computer Science and Applications 8/10, 2017.

[An18] Androulaki, E.; Barger, A.; Bortnikov, V.; Cachin, C.; Christidis, K.; De Caro, A.; Enyeart, D.; Ferris, C.; Laventman, G.; Manevich, Y., et al.: Hyperledger fabric: a distributed operating system for permissioned blockchains. In: Proceedings of the thirteenth EuroSys conference. Pp. 1–15, 2018.

[Ay22] Ayala, D.; Hernández, I.; Ruiz, D.; Rahm, E.: Leapme: Learning-based property matching with embeddings. Data & Knowledge Engineering 137/, p. 101943, 2022.

[Ba16] Basu, D. et al.: Regularized cost-model oblivious database tuning with reinforcement learning. Trans. Large-Scale Data-and Knowledge-Centered Systems XXVIII/, pp. 96–132, 2016.

[BD11] Bellahsene, Z.; Duchateau, F.: Tuning for schema matching. Schema Matching and Mapping/, pp. 293–316, 2011.

[Be19] Berrar, D.: Performance Measures for Binary Classification. 2019.

[BMR11] Bernstein, P. A.; Madhavan, J.; Rahm, E.: Generic schema matching, ten years later. Proc. VLDB Endowment 4/11, pp. 695–701, 2011.

[Br97] Bradley, A. P.: The use of the area under the ROC curve in the evaluation of machine learning algorithms. Pattern recognition 30/7, pp. 1145–1159, 1997.

[CJ07] Calders, T.; Jaroszewicz, S.: Efficient AUC optimization for classification. In: European conference on principles of data mining and knowledge discovery. Springer, pp. 42–53, 2007.

[CJ20] Chicco, D.; Jurman, G.: The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC genomics 21/, pp. 1–13, 2020.

[CJ23] Chicco, D.; Jurman, G.: The Matthews correlation coefficient (MCC) should replace the ROC AUC as the standard metric for assessing binary classification. BioData Mining 16/1, p. 4, 2023.

[DG24] De La Fuente, N.; Guerra, D. A. V.: A Comparative Study of Deep Reinforcement Learning Models: DQN vs PPO vs A2C. arXiv preprint arXiv:2407.14151/, 2024.

[DR02] Do, H.-H.; Rahm, E.: COMA: a system for flexible combination of schema matching approaches. In: Proc. 28th Int. Conf. on Very Large Data Bases. VLDB Endowment, pp. 610–621, 2002.

[Fa13] Faria, D.; Pesquita, C.; Santos, E.; Palmonari, M.; Cruz, I. F.; Couto, F. M.: The AgreementmakerLight ontology matching system. In: Proc. OTM. Springer, pp. 527–541, 2013.

[Fa14] Faria, D.; Pesquita, C.; Santos, E.; Cruz, I. F.; Couto, F. M.: AgreementMakerLight 2.0: towards efficient large-scale ontology matching. In: ISWC (Posters & Demos). CEUR-WS.org, Aachen, DEU, pp. 457–460, 2014.

[Fa19] Faria, D.; Pesquita, C.; Tervo, T.; Couto, F. M.; Cruz, I. F.: AML and AMLC results for OAEI 2019. In: Proc. 14th Int. Workshop on Ontology Matching. Vol. 2536, pp. 101–106, 2019.

[Fe10] Fellbaum, C.: WordNet. In: Theory and applications of ontology: computer applications. Springer, pp. 231–243, 2010.

[Fe18] Fernandez, R. C. et al.: Seeping semantics: Linking datasets using word embeddings for data discovery. In: 2018 IEEE 34th Int. Conf. on Data Engineering (ICDE). IEEE, pp. 989–1000, 2018.

[FZY10] Feng, Y.; Zhao, L.; Yang, J.: GATuner: Tuning schema matching systems using genetic algorithms. In: 2010 2nd International Workshop on Database Technology and Applications. IEEE, pp. 1–4, 2010.

[HO20] Huang, S.; Ontañón, S.: A closer look at invalid action masking in policy gradient algorithms. arXiv preprint arXiv:2006.14171/, 2020.

[Ho24] Hofer, M.; Obraczka, D.; Saeedi, A.; Köpcke, H.; Rahm, E.: Construction of Knowledge Graphs: Current State and Challenges. Information 15/8, 2024.

[HP23] Hertling, S.; Paulheim, H.: Olala: Ontology matching with large language models. In: Proceedings of the 12th Knowledge Capture Conference 2023. Pp. 131–139, 2023.

[HZZ22] Huang, X.; Zhang, H.; Zhai, X.: A Novel Reinforcement Learning Approach for Spark Configuration Parameter Optimization. Sensors 22/15, p. 5930, 2022.

[JC11] Jiménez-Ruiz, E.; Cuenca Grau, B.: Logmap: Logic-based and scalable ontology matching. In: Proc. Int. Semantic Web Conference (ISWC). Springer, pp. 273–288, 2011.

[Ji12] Jiménez-Ruiz, E.; Cuenca Grau, B.; Zhou, Y.; Horrocks, I.: Large-scale interactive ontology matching: Algorithms and implementation. In: ECAI 2012. Ios Press, pp. 444–449, 2012.

[KKK18] Kolyvakis, P.; Kalousis, A.; Kiritsis, D.: DeepAlignment: Unsupervised ontology matching with refined word vectors. In: Proc. 16th Annual Conf. of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 1-6 June 2018, 2018.

[Ko21a] Koutras, C.; Psarakis, K.; Siachamis, G.; Ionescu, A.; Fragkoulis, M.; Bonifati, A.; Katsifodimos, A.: Valentine in action: matching tabular data at scale. Proceedings of the VLDB Endowment 14/12, pp. 2871–2874, 2021.

[Ko21b] Koutras, C.; Siachamis, G.; Ionescu, A.; Psarakis, K.; Brons, J.; Fragkoulis, M.; Lofi, C.; Bonifati, A.; Katsifodimos, A.: Valentine: Evaluating matching techniques for dataset discovery. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, pp. 468–479, 2021.

[Le07] Lee, Y.; Sayyadian, M.; Doan, A.; Rosenthal, A. S.: eTuner: tuning schema matching software using synthetic scenarios. The VLDB journal 16/, pp. 97–122, 2007.

[Li18] Liang, E. et al.: RLlib: Abstractions for Distributed Reinforcement Learning. In: Proc. 35th Int.Conf. on Machine Learning. Vol. 80, PMLR, pp. 3053–3062, 2018.

[Li19] Li, G.; Zhou, X.; Li, S.; Gao, B.: Qtune: A query-aware database tuning system with deep reinforcement learning. Proc. VLDB Endowment 12/12, pp. 2118–2130, 2019.

[Li23] Li, M.; Wang, Y.; Ma, S.; Liu, C.; Huo, D.; Wang, Y.; Xu, Z.: Auto-Tuning with Reinforcement Learning for Permissioned Blockchain Systems. Proc. VLDB Endowment 16/5, pp. 1000–1012, 2023.

[Ma11] Massmann, S.; Raunich, S.; Aumüller, D.; Arnold, P.; Rahm, E.: Evolution of the COMA match system. Ontology Matching 49/, pp. 49–60, 2011.

[Mn16] Mnih, V.: Asynchronous Methods for Deep Reinforcement Learning. arXiv preprint arXiv:1602.01783/, 2016.

[MSH09] Motik, B.; Shearer, R.; Horrocks, I.: Hypertableau reasoning for description logics. Journal of Artificial Intelligence Research 36/, pp. 165–228, 2009.

[On23a] Ontology Alignment Evaluation Initiative: The anatomy real world case, 2023, URL: https://oaei.ontologymatching.org/2023/results/anatomy/index.html, visited on: 12/12/2024.

[On23b] Ontology Alignment Evaluation Initiative: The anatomy real world case, 2023, URL: https://oaei.ontologymatching.org/2023/anatomy/index.html, visited on: 12/12/2024.

[OYD21] Osman, I.; Yahia, S. B.; Diallo, G.: Ontology integration: approaches and challenging issues. Information Fusion 71/, pp. 38–63, 2021.

[Pa24] Parciak, M.; Vandevoort, B.; Neven, F.; Peeters, L. M.; Vansummeren, S.: Schema Matching with Large Language Models: an Experimental Study. arXiv preprint arXiv:2407.11852/, 2024.

[Pe11] Pedregosa, F. et al.: Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research 12/, pp. 2825–2830, 2011.

[PER12] Peukert, E.; Eberius, J.; Rahm, E.: A self-configuring schema matching system. In: 2012 IEEE 28th International Conference on Data Engineering. IEEE, pp. 306–317, 2012.

[Po21] Po, M. N. et al.: Results of the Ontology Alignment Evaluation Initiative 2021. In: Proc. 16th Int. Workshop on Ontology Matching. Vol. 3063. CEUR Workshop Proceedings, pp. 62–108, 2021.

[Ra11] Rahm, E.: Towards Large-Scale Schema and Ontology Matching. In (Bellahsene, Z.; Bonifati, A.; Rahm, E., eds.): Schema Matching and Mapping. Springer, pp. 3–27, 2011.

[Ra16] Rahm, E.: The Case for Holistic Data Integration. In: Advances in Databases and Information Systems. Springer, pp. 11–27, 2016.

[RB01] Rahm, E.; Bernstein, P. A.: A survey of approaches to automatic schema matching. VLDB Journal 10/4, pp. 334–350, 2001.

[Sc17] Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O.: Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347/, 2017.

[SCA17] Schulman, J.; Chen, X.; Abbeel, P.: Equivalence between policy gradients and soft q-learning. arXiv preprint arXiv:1704.06440/, 2017.

[SGR20] Shraga, R.; Gal, A.; Roitman, H.: Adnev: Cross-domain schema matching using deep similarity matrix adjustment and evaluation. Proc. VLDB Endowment 13/9, pp. 1401–1415, 2020.

[SKH15] Simancik, F.; Kazakov, Y.; Horrocks, I.: Consequence- Based Reasoning beyond Horn Ontologies./, 2015.

[SS24] Seedat, N.; van der Schaar, M.: Matchmaker: Self-Improving Large Language Model Programs for Schema Matching. arXiv preprint arXiv:2410.24105/, 2024.

[Ta20] Tang, C.-Y.; Liu, C.-H.; Chen, W.-K.; You, S. D.: Implementing action mask in proximal policy optimization (PPO) algorithm. ICT Express 6/3, pp. 200–203, 2020.

[To23] Towers, M. et al.: Gymnasium, 2023, URL: https://zenodo.org/record/8127025, visited on: 07/08/2023.

[VC18] Vaquero, L. M.; Cuadrado, F.: Auto-tuning distributed stream processing systems using reinforcement learning. arXiv preprint arXiv:1809.05495/, 2018.

[We23] Web Data Commons: WDC Schema Matching Benchmark, 2023, URL: https://webdatacommons.org/structureddata/smb, visited on: 12/12/2024.

[Zh19] Zhang, J. et al.: An End-to-End Automatic Cloud Database Tuning System Using Deep Reinforcement Learning. In: Proc. ACM SIGMOD conf. Pp. 415–432, 2019.