# Porting ChatKBQA to Wikidata

**Submitted to the**
**Albert-Ludwigs-Universität Freiburg**

**by Luis Josef Drayer**

21. September 2026

universität freiburg

# Bachelor's Thesis

Submitted on **September 21, 2026**
to the **Albert-Ludwigs-Universität Freiburg**
by **Luis Josef Drayer** (5542256)
Examined by **Prof. Dr. Hannah Bast**

Chair for Algorithms and Data Structures
Department of Computer Science
Faculty of Engineering
Albert-Ludwigs-Universität Freiburg

# Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used.

Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.

I also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Freiburg, September 21, 2026

Place, date

Luis Drayer

Luis Josef Drayer

# Declaration on the Use of AI

I hereby declare that I used generative AI tools in a limited and assistive manner in my thesis. Their use was restricted to language support, grammar and spelling correction, readability improvements, formatting assistance, support in structuring or refining text, and assistance in literature search. The tools were not used to replace my own scientific work, technical decisions, analysis, interpretation, or conclusions. I reviewed and verified all AI-assisted content. All scientific content, implementation review, evaluation decisions, interpretations, and conclusions remain my own responsibility.

# Abstract

ChatKBQA is a generate-then-retrieve method for knowledge graph question answering that was originally developed and evaluated on Freebase. In this work, we extend ChatKBQA by separating entity and predicate retrieval into a modular system of configurable linkers and encapsulating knowledge-base-dependent functionality in dedicated Knowledge Base modules. This allows the retrieval process to be adapted to different knowledge graphs without modifying the core pipeline. Using this infrastructure, we reproduce the original ChatKBQA approach and transfer it from Freebase to Wikidata. We evaluate the resulting system on several established Wikidata benchmarks and investigate the effects of the retrieval configuration and training representation.

# Zusammenfassung

ChatKBQA ist eine Generate-then-Retrieve-Methode zur Beantwortung von Fragen über Wissensgraphen, die ursprünglich für Freebase entwickelt und evaluiert wurde. In dieser Arbeit erweitern wir ChatKBQA, indem wir Entity- und Predicate-Retrieval in ein modulares System konfigurierbarer Linker aufteilen und wissensgraph-spezifische Funktionalität in eigene Knowledge-Base-Module auslagern. Dadurch kann der Retrieval-Schritt an verschiedene Wissensgraphen angepasst werden, ohne die Kernpipeline verändern zu müssen. Auf Grundlage dieser Infrastruktur reproduzieren wir den ursprünglichen ChatKBQA-Ansatz und übertragen ihn von Freebase auf Wikidata. Das resultierende System wird auf mehreren etablierten Wikidata-Benchmarks evaluiert, wobei insbesondere der Einfluss der Retrieval-Konfiguration und der Trainingsrepräsentation untersucht wird.

# Contents


**1 Introduction** **1**
- 1.1 Motivation . . . 1
- 1.2 Contribution . . . 1
- 1.3 Related Work . . . 2

**2 Background** **3**
- 2.1 RDF . . . 3
- 2.2 SPARQL . . . 4
- 2.3 Knowledge Bases . . . 5
  - 2.3.1 Freebase . . . 5
  - 2.3.2 Wikidata . . . 5
  - 2.3.3 SPARQL Engines . . . 5
- 2.4 Apache Jena . . . 5
- 2.5 FACC1 . . . 6
- 2.6 Large Language Models . . . 6
- 2.7 SimCSE Similarity . . . 6
- 2.8 Low-Rank Adaptation (LoRA) . . . 6
- 2.9 LLaMA-Factory . . . 7

**3 Methodology** **8**
- 3.1 Overview of ChatKBQA . . . 8
- 3.2 Training Format Conversion . . . 8
  - 3.2.1 Original Approach . . . 8
  - 3.2.2 Extension . . . 10
  - 3.2.3 Gold Entity and Predicate Maps . . . 12
  - 3.2.4 Type Map . . . 12
- 3.3 Fine-Tuning LLM and Beam Search Inference . . . 12
- 3.4 Unsupervised Retrieval for Entities and Predicates . . . 13
  - 3.4.1 Original Approach . . . 13
  - 3.4.2 Extension . . . 17
- 3.5 Evaluation . . . 20
  - 3.5.1 Original Approach . . . 20
  - 3.5.2 Extension . . . 21

**4 Implementation Details** **23**
- 4.1 Entity and Predicate Linkers . . . 23
  - 4.1.1 Definition . . . 23
  - 4.1.2 ChatKBQA Entity Linkers . . . 24
  - 4.1.3 ChatKBQA Predicate Linkers . . . 25

4.1.4 Wikidata Entity Linkers . . . 26
4.1.5 Wikidata Predicate Linkers . . . 28
4.2 Knowledge Base Module . . . 29
4.3 Complexity Analysis . . . 31

**5 Results** **33**
5.1 Hardware and Software Configuration . . . 33
5.2 Validation of the Extended Implementation . . . 33
5.3 Retraining of the Original Models . . . 35
5.4 Hyperparameter Analysis . . . 37
5.4.1 ChatKBQA Hyperparameter Configuration . . . 37
5.4.2 Runtime and Performance per Retrieval Pass . . . 39
5.4.3 Winning Query Candidate Distributions . . . 41
5.4.4 Hyperparameter sensitivity . . . 42
5.4.5 Retrieval Parameter Reduction . . . 44
5.5 Further experiments . . . 45
5.5.1 Training Format Comparison . . . 45
5.5.2 Experiments on Wikidata . . . 47
5.5.3 Runtime and Performance of Wikidata experiments . . . 47
5.5.4 Performance comparison to GRISP . . . 49
5.6 Error Analysis . . . 50

**6 Discussion** **52**
6.1 Reproduction of ChatKBQA . . . 52
6.2 Retrieval Search Space . . . 52
6.3 SPARQL and Jena Representations . . . 53
6.4 Transfer to Wikidata . . . 53
6.5 Comparison with GRISP . . . 54
6.6 Generation and Retrieval . . . 54
6.7 Limitations and Future Work . . . 55

**7 Conclusion** **57**

**List of Figures** **II**

**List of Tables** **III**

**List of Algorithms** **IV**

**Glossary** **V**

**References** **VI**

**Appendix** **XI**

**A Additional Results** **XI**

A.1 Impact of label fallback on metrics . . . . . XI

A.2 Full winning index distribution in retrain experiments . . . . . XII

A.2.1 Simple . . . . . XII

A.2.2 Neighborhood . . . . . XIII

**B Training Configuration** **XIII**

**C Benchmark Split Size** **XV**

# 1 Introduction

While the internet contains vast amounts of information, much of it is not available in a structured manner. Knowledge graphs aim to provide a structured representation of knowledge. The Resource Description Framework (RDF), for example, provides a standardized data model to represent relationships between resources. SPARQL, a query language for RDF, can then be used for structured queries over this information. An increasing number of knowledge graphs are available in RDF. This includes both broad-domain knowledge graphs like Wikidata [25] and Freebase [6], as well as more domain-specific knowledge graphs like UniProt [7] or OpenStreetMap [16]. With the growing availability of structured knowledge, knowledge graph question-answering (KGQA) has become an important research area. While KGQA refers to knowledge graph question-answering over any kind of knowledge graph, this work focuses on RDF-based knowledge graphs. KGQA methods aim to answer natural-language questions by generating an executable SPARQL query. This SPARQL query is then used to retrieve an answer from a knowledge graph.

One such KGQA method is ChatKBQA [15]. It relies on a large language model (LLM) to generate an intermediate logical form. This logical form is then processed by unsupervised entity and predicate retrieval in order to produce an executable SPARQL query.

## 1.1 Motivation

The original ChatKBQA implementation is tightly coupled to the WebQuestionsSP (WebQSP) [31] and ComplexWebQuestions (CWQ) [20] datasets, as well as the Freebase knowledge graph. Its data processing, logical-form generation, entity and predicate retrieval, and evaluation components are designed specifically around these datasets. This limits the applicability of the original implementation as a performance baseline when evaluating KGQA methods on other datasets or knowledge graphs. This is especially relevant given that Freebase is no longer actively maintained, while Wikidata provides a widely used alternative RDF knowledge graph.

## 1.2 Contribution

This work extends the ChatKBQA method in two concrete ways. First, we generalize ChatKBQA to support different datasets and knowledge graphs rather than being tied to WebQSP and CWQ. Second, we separate the candidate-generation and execution process from the concrete entity and predicate retrieval mechanisms of the original. The retrieval mechanisms are instead extracted into modular linkers. This modular

design allows knowledge-graph- or dataset-specific retrieval strategies to be defined independently of the core pipeline.

## 1.3 Related Work

Existing KGQA methods can broadly be divided into retrieval-based, semantic-parsing-based and agentic approaches.

**Retrieval- or Exploration-based methods** Methods in this category place more importance on direct knowledge graph exploration or restricting the search space. For example, Think-On-Graph [19] explores the knowledge graph by traversing the graph's edges to derive an answer based on the retrieved information. Other approaches make use of Subgraph Retrieval [32]. This restricts the search space from the entire knowledge graph to a smaller subgraph that is deemed relevant to the question. The resulting subgraph can then be combined with a reasoning path generator to further explore the subgraph.

**Semantic-parsing-based methods** Semantic-parsing-based methods translate a natural-language question into a structured query representation, which can subsequently be executed against the knowledge graph itself to obtain an answer. WikiSP [30] is a fine-tuning-based method that follows a retrieve-then-generate approach. It uses ReFinEd [3] to extract relevant Wikidata entity identifiers directly from the natural language question. These identifiers are then provided to the LLM when generating the SPARQL query. Identifiers that are not directly provided by ReFinEd are generated using their label. These are resolved to identifiers in a later step. GRISP [27] is, much like ChatKBQA, a fine-tuning-based method of the generate-then-retrieve kind. It generates a set of SPARQL skeletons with placeholders for entities and predicates. The language model is fine-tuned both to generate these skeletons and to re-rank candidate entities and predicates for their respective placeholders. The candidates for each placeholder are iteratively re-ranked until an executable SPARQL query is obtained. Notably, GRISP features a backtracking mechanism if the model deems no candidate to be fit for a placeholder. ChatKBQA [15] is a fine-tuning-based generate-then-retrieve method and forms the basis of this work. It uses a fine-tuned LLM to generate an intermediate Logical Form, after which entities and predicates are retrieved to produce an executable SPARQL query.

**Agentic methods** GRASP [26] is an agentic zero-shot method. A large language model interacts iteratively with the knowledge graph by calling pre-defined functions to answer a question.

# 2 Background

In this section, we present the concepts that are relevant to understand the approach and implementation.

## 2.1 RDF

The Resource Description Framework (RDF) [29] defines a standardized and structured data model. At its core, RDF represents knowledge as a set of triples. These triples each have a subject, predicate and object. A single triple resembles a simple statement in natural language. Figure 1 shows the statement ‘Freiburg is a city.‘ visualized as an RDF triple.

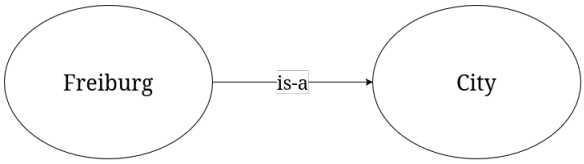



**Figure 1:** Simple representation of a single RDF triple.

In practice, an RDF graph can contain a large number of triples. These triples can then be represented as a directed graph with labeled edges. The subjects and objects of triples represent vertices, while predicates represent a directed, labeled edge. A resource can be identified by an Internationalized Resource Identifier (IRI) [29], while concrete values such as strings, numbers or dates can be represented as literals. IRIs are especially important when distinguishing between two different vertices that share the same natural language name. A possibly denser subgraph is shown in Figure 2.

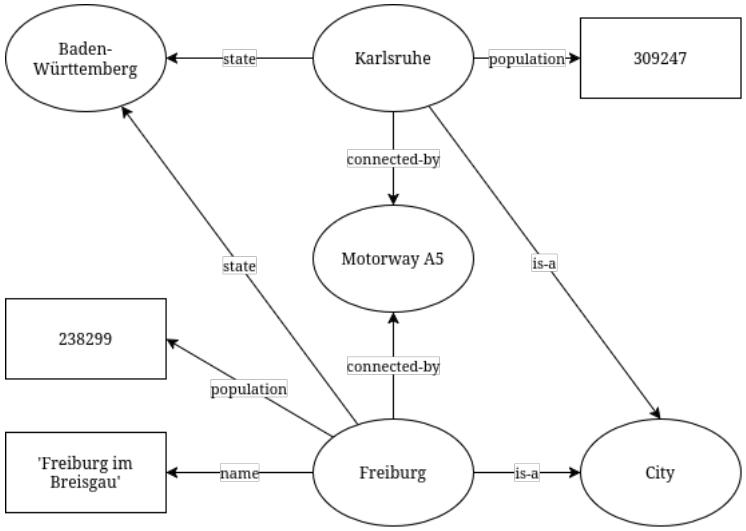



**Figure 2:** A possible RDF knowledge graph. Square vertices indicate literals, while oval vertices indicate IRIs.

## 2.2 SPARQL

SPARQL Protocol and RDF query language (SPARQL) is a query language for RDF graphs. Its semantics and syntax are defined by the World Wide Web Consortium (W3C) [12].

A common form of SPARQL query is a `SELECT` query. A `SELECT` query specifies which variables should be returned, while the accompanying `WHERE` clause defines the graph pattern that must be matched in the RDF graph. To extract all cities from the abstract graph shown in Figure 2, the SPARQL query shown in Code 1 can be used.

```
PREFIX ex_ent: <http://rdf.example.com/entity/>
PREFIX ex_pre: <https://rdf.example.com/predicate/>

SELECT ?x ?name
WHERE {
    ?x ex_pre:is-a ex_ent:City .
    OPTIONAL {
        ?x ex_pre:name ?name .
    }
}
```

**Code 1:** Abstract SPARQL to retrieve all entities of type city.

The results this query would return are shown in Table 1.

| ?x | ?name |
|---|---|
| http://rdf.example.com/entity/Freiburg | 'Freiburg im Breisgau' |
| http://rdf.example.com/entity/Karlsruhe | |

**Table 1:** Result of SPARQL query defined in Code 1 if it were executed against the RDF graph shown in Figure 2.

The query returns both cities defined in the graph. However, only the entity `Freiburg` has a triple containing the predicate `name`. Because the corresponding triple pattern in the SPARQL query is defined as `OPTIONAL`, the entity `Karlsruhe` is included in the results despite this.

## 2.3 Knowledge Bases

This section briefly explains the concrete knowledge graphs used in this work. Additionally, the SPARQL engines used in this work to interact with these knowledge graphs are introduced. While the terms knowledge graph and knowledge base can have different meanings in the literature, they are used interchangeably throughout this work to refer to the underlying knowledge graph queried by the system.

### 2.3.1 Freebase

Freebase is a broad-domain knowledge base that was developed as a collaborative database of structured information [6]. Development of Freebase was discontinued in 2014, and the service was subsequently made read-only before being shut down. Despite this, Freebase remains relevant to KGQA research because several established datasets and methods are based on it. Notably, this includes ChatKBQA [15].

### 2.3.2 Wikidata

Wikidata is, like Freebase, a broad-domain knowledge base [25]. It features more than 120 million items [28]. It is continuously maintained and expanded.

### 2.3.3 SPARQL Engines

**Virtuoso** Virtuoso is a database system that provides support for RDF data and the SPARQL query language [17]. ChatKBQA uses Virtuoso as its SPARQL engine. To ensure a comparable environment, experiments on Freebase in this work use the same Virtuoso setup.

**QLever** QLever is an RDF graph database that implements the RDF and SPARQL standards [4]. In this work, QLever is used as the SPARQL engine for executing queries against Wikidata. QLever was selected based on its reported performance compared to other SPARQL engines [5].

## 2.4 Apache Jena

Apache Jena is an open-source framework for working with RDF data and SPARQL [1]. While it also provides a SPARQL engine, this work does not use it for query execution. Instead, the SPARQL parser included in Apache Jena is used to convert the original SPARQL queries of a given dataset into one of two training formats. More details, along with examples, are provided in Section 3.2.2.

## 2.5 FACC1

The Freebase Annotation of the ClueWeb Corpora (FACC1) is a collection of annotations linking entity mentions in the ClueWeb corpus to corresponding Freebase entities [10]. The annotations associate textual mentions with Freebase entity identifiers and confidence scores. In ChatKBQA, and by extension this work as well, a surface index derived from FACC1 annotations is used for entity retrieval. Given a textual surface form, the index provides candidate Freebase entities together with a surface score indicating the strength of the match.

## 2.6 Large Language Models

Large Language Models (LLMs) are neural language models trained on large collections of text to learn statistical patterns in natural language. Modern LLMs are commonly based on the Transformer architecture [24] and can generate text by predicting the next token given the preceding context. The models used in this work are Llama 2 [21] and Qwen2.5 [18].

## 2.7 SimCSE Similarity

SimCSE [11] is a framework for learning sentence embeddings that can be used to measure semantic similarity between different text sequences. It is based on contrastive learning, where semantically related representations are encouraged to be closer in the embedding space. Given two text sequences, a pretrained SimCSE model can encode them into vector representations. Their semantic similarity can then be estimated using the cosine similarity between the resulting vectors. In this work, SimCSE embeddings are used during retrieval to compare entity and predicate mentions in the prediction beam to a set of retrieved candidates.

## 2.8 Low-Rank Adaptation (LoRA)

Low-Rank Adaptation (LoRA) [13] is a parameter-efficient fine-tuning method for adapting pretrained neural networks to specific tasks. Instead of updating the parameters of the entire model, LoRA adds trainable low-rank matrices to selected layers while keeping the original model parameters frozen. This substantially reduces the number of trainable parameters and the memory requirements during fine-tuning. The resulting LoRA adapter can subsequently be used together with the original model to perform inference with the adapted model. In this work, LoRA is used to fine-tune the language models on the training data.

## 2.9 LLaMA-Factory

LLaMA-Factory [33] is an open-source framework for fine-tuning and inference of large language models. It provides implementations for various parameter-efficient fine-tuning methods, including LoRA, and supports a range of pretrained language models. In this work, LLaMA-Factory is used to fine-tune the language models using LoRA. It supersedes the LLMTuner framework used by the original ChatKBQA implementation.

# 3 Methodology

In this section, we describe the ChatKBQA method in detail. We first present the general approach and then describe the differences between the original implementation and our extension.

## 3.1 Overview of ChatKBQA

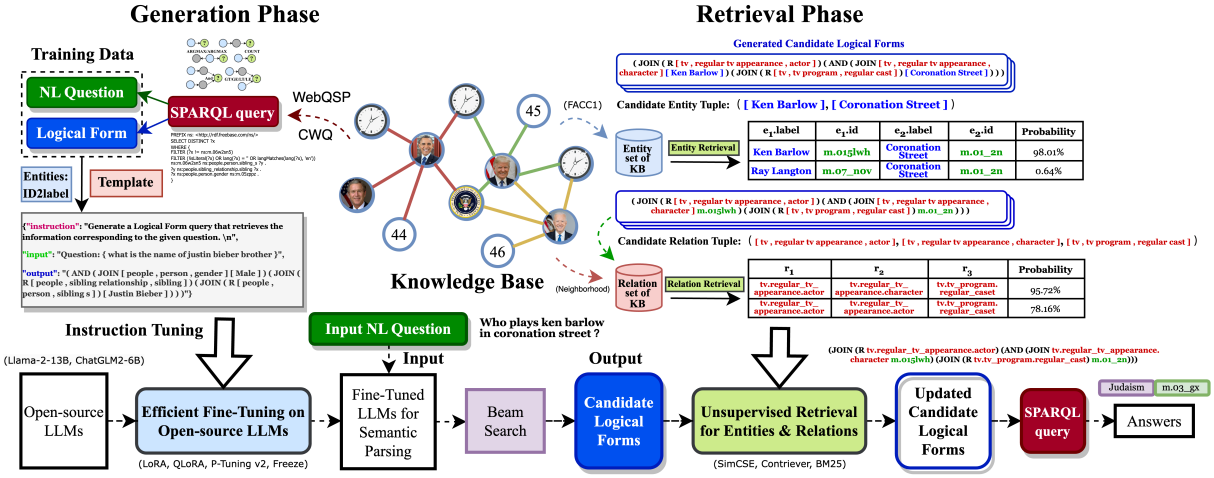



**Figure 3:** Overview of the ChatKBQA architecture [15]

As shown in Figure 3, ChatKBQA can be separated into two phases:

- Generation Phase
  - SPARQL → Training Format Conversion
  - Fine-Tuning LLM and Beam Search inference
- Retrieval Phase
  - Unsupervised Retrieval for Entities and Predicates

## 3.2 Training Format Conversion

### 3.2.1 Original Approach

While the final output of ChatKBQA's Retrieval Algorithm (described later in Section 3.4.1) for a single dataset item is a SPARQL query, so-called Logical Forms are used as an intermediate representation during training [15]. They provide a more structured and compositional representation of the query. This reduces the syntactic complexity that the model has to reproduce during training. Take the query in Code 2 for example, which answers the following question: `What is the name of Justin Bieber's brother?`

```
PREFIX ns: <http://rdf.freebase.com/ns/>
SELECT DISTINCT ?x
WHERE {
    FILTER (?x != ns:m.06w2sn5)
    FILTER (!isLiteral(?x) OR lang(?x) =
    '' OR langMatches(lang(?x), 'en'))
    ns:m.06w2sn5 ns:people.person.sibling_s ?y .
    ?y ns:people.sibling_relationship.sibling ?x .
    ?x ns:people.person.gender ns:m.05zppz .
}
```

**Code 2:** Example SPARQL query taken from WebQSP [31]

This query is transformed into the Logical Form shown in Code 3:

```
( AND
  ( JOIN
    [ people , person , gender ]
    [ Male ]
  )
  ( JOIN
    ( R [ people , sibling relationship , sibling ] )
    ( JOIN
      ( R [ people , person , sibling s ] )
      [ Justin Bieber ]
    )
  )
)
```

**Code 3:** Example ChatKBQA Logical Form

This Logical Form is obtained by taking the triples of the original SPARQL query and arranging them into a logical chain, using the syntax listed in Table 2. Afterwards, all entity and predicate identifiers are replaced by their corresponding natural language labels. This removes dependencies on knowledge base-specific IDs and allows the model to generate logical forms directly from natural language questions.

| Operator | Syntax | Description |
|---|---|---|
| AND | (AND E1 E2) | Computes the intersection of the results of E1 and E2. |
| JOIN | (JOIN E1 r) | Applies predicate r to the entities obtained from E1, returning the connected entities through that predicate. |
| R | (R r) | Represents the reverse direction of predicate r, retrieving entities that point to the input entity through that predicate. |
| COUNT | (COUNT E1) | Returns the number of results in E1. |
| ARGMAX | (ARGMAX E1 r) | Returns the entity in E1 with the maximum value obtained by projecting predicate r. |
| ARGMIN | (ARGMIN E1 r) | Returns the entity in E1 with the minimum value obtained by projecting predicate r. |
| GT | (GT E1 l) | Restricts E1 to values greater than literal l. |
| GE | (GE E1 l) | Restricts E1 to values greater than or equal to literal l. |
| LT | (LT E1 l) | Restricts E1 to values less than literal l. |
| LE | (LE E1 l) | Restricts E1 to values less than or equal to literal l. |

**Table 2:** Syntax of the Logical Form representation used by the original ChatKBQA approach.

### 3.2.2 Extension

While Logical Forms reduce the syntactical complexity of SPARQL queries and provide a structured representation for model training, they are also limited in the range of queries they can express. The tree-based structure is designed around a fixed set of operators and does not cover the full expressiveness of SPARQL. One simple example is a query with more than one SELECT variable. Since Logical Forms represent the result of a query as a single entity retrieval operation, they cannot naturally represent queries that return tuples of multiple variables. Because of these limitations, our extension moves away from the original Logical Form syntax. Instead, we use Apache Jena (see Section 2.4) to generate two alternative training formats that fully preserve the expressiveness of the underlying SPARQL representation:

- Jena Syntax Expression (S-Expression)
- SPARQL Query representation

**Jena Syntax Expression** The first representation uses the so-called Syntax Expressions [2] generated by Apache Jena. Unlike Logical Forms, a Syntax Expression represents the complete structure of a SPARQL query, including operations such as

filters, joins, projections, and aggregations. While this is closer in spirit to the Logical Forms, it is also verbose and abstract. Code 4 shows the query of Code 2 converted to a Syntax Expression using Apache Jena.

```
(distinct
  (project (?x)
    (filter
      (exprlist
        (!= ?x fb:Justin_Bieber)
        (||
          (|| (! (isLiteral ?x)) (= (lang ?x) ""))
          (langMatches(lang ?x) "en")
        )
      )
      (bgp
        (triple fb:Justin_Bieber fbp:people.person.sibling_s ?y)
        (triple ?y fbp:people.sibling_relationship.sibling ?x)
        (triple ?x fbp:people.person.gender fb:Male)
      )
    )
  )
)
```

**Code 4:** Example Jena Syntax Expression

**SPARQL representation** The second representation directly uses the SPARQL query as the training target. However, this does not simply take the original SPARQL query from the dataset. Instead, the dataset item's SPARQL Query is first converted to a Syntax Expression using Apache Jena. Afterwards, the resulting Syntax Expression is converted back into a SPARQL query, utilizing Apache Jena once more. The Syntax Expression of Code 4 converted into a normalized SPARQL query can be seen in Code 5.

```
SELECT DISTINCT  ?x
WHERE
  { fb:Justin_Bieber
              fbp:people.person.sibling_s  ?y .
    ?y        fbp:people.sibling_relationship.sibling  ?x .
    ?x        fbp:people.person.gender  fb:Male
    FILTER ( ?x != fb:Justin_Bieber )
    FILTER ( ( ( ! isLiteral(?x) ) || ( lang(?x) = "" ) ) ||
             langMatches(lang(?x), "en") )
  }
```

**Code 5:** Example of a normalized SPARQL query

Using this roundtrip, we get a much more uniform representation of SPARQL queries, that is even consistent in style across different datasets' SPARQL format inconsistencies. This is because Apache Jena normalizes the style of the triples and filters, even though they might look very different across different datasets.

### 3.2.3 Gold Entity and Predicate Maps

During logical form generation, when entity and predicate identifiers are substituted for their corresponding labels, those transformations are written to a *gold entity map* and *gold predicate map* respectively. This information can later be used during retrieval to isolate the impact of certain components on the overall performance of the method.

### 3.2.4 Type Map

In addition to the gold entity and relation maps, a global *Type Map* is constructed during data preprocessing. For every entity occurrence in the training data, the knowledge base is queried to determine whether the entity also appears as a type (i.e., whether it is used as a type definition for other entities). If so, the entity identifier is recorded in the type map along with its label. For example, some SPARQL query in the training data could contain the entity `ns:m.01mp` (Country). In Freebase, this entity is also defined as a type. This means that other entities can be of type "Country". This entity then becomes an entry in the global type map for this dataset, as shown in Code 6. This type map is later passed as additional information to the retrieval algorithm.

```
{
    ...
    "ns:m.01mp": "Country",
    ...
}
```

**Code 6:** Type map entry for the Country entity

## 3.3 Fine-Tuning LLM and Beam Search Inference

The original ChatKBQA implementation used LLMTuner (see Section 2.9) for both model fine-tuning and inference. In the extended version, LLMTuner was replaced with LLaMA-Factory, its successor and the currently maintained framework for large language model fine-tuning. This change primarily affects the underlying training infrastructure, while the overall fine-tuning procedure and inference strategy remain unchanged. The fine-tuning process follows a parameter-efficient approach using Low-Rank Adaptation (see Section 2.8).

For inference, we use the group beam search implementation provided by the transformers-community/group-beam-search plugin [14]. Compared to regular beam search, this produces more distinct beams across beam groups. This is because of a diversity penalty that encourages more diverse outputs. This inference strategy mirrors the original ChatKBQA approach, which employed the same group beam search mechanism via LLMTuner's `chat_beam` method. An example of a group beam search inference output can be seen in Code 7. Note that, as this model was trained on WebQSP, the `FILTER` block's structure is always the same. This is because every item in WebQSP uses this minimal filter.

```
1. SELECT DISTINCT ?x WHERE { fb:Jamaica fbp:location.country.
   languages_spoken ?x
   FILTER (?x != fb:Jamaica) FILTER ((!isLiteral(?x) || lang(?x) = "")
      || langMatches(lang(?x), "en")) }

2. Select * WHERE { fb:Jamaica fbp:location.country.languages_spoken ?x
   FILTER (?x != fb:Jamaica) FILTER ((!isLiteral(?x) || lang(?x) = "")
      || langMatches(lang(?x), "en")) }

3. SELECT DISTINCT ?x WHERE { fb:Jamaica fbp:location.country.
   official_language ?x
   FILTER (?x != fb:Jamaica) FILTER ((!isLiteral(?x) || lang(?x) = "")
      || langMatches(lang(?x), "en")) }

4. SELECT DISTINCT ?x WHERE { fb: Jamaica fbp: location.country.
   official_language ?x
   FILTER (?x != fb:Jamaica) FILTER ((!isLiteral(?x) || lang(?x) = "")
      || langMatches(lang(?x), "en")) }

5. SELECT DISTINCT ?x WHERE { fb:Jamacia fbp:location.Country.
   Languages_Spoken ?x
   FILTER (?x != fb:Jamacia) FILTER ((!isLiteral(?x) || lang(?x) = "")
      || langMatches(lang(?x), "en")) }
```

**Code 7:** Sample group beam search inference output (5 candidate SPARQL queries)

While some of the beams for this simple query differ only in capitalization, others show meaningful variation in the predicted predicate, as seen in beams 1 and 3.

## 3.4 Unsupervised Retrieval for Entities and Predicates

### 3.4.1 Original Approach

The original Retrieval Algorithm follows a strict order of operations. Algorithm 1 describes the approach on a per-item basis.

**Algorithmus 1 :** Unsupervised Retrieval for Entities and Predicates (ChatKBQA)

**Input :** Prediction beams $B$, global type map $T$, gold entity flag $G$, gold entity map $M$, top-$k$ thresholds $k_e, k_p$, similarity thresholds $t_{e\text{-gold}}, t_e, t_p$, number of entity permutations considered for predicate retrieval $n_e$, number of predicate permutations tried $n_p$

**Output :** successful SPARQL query $q^*$, successful logical form $l^*$, query result $r^*$

**Function** `RetrieveEntities`$(b, T, G, M, k_e, t_{e\text{-gold}}, t_e)$**:**

- **foreach** *entity mention* $e \in b$ **do**
  - $C_e \leftarrow \emptyset$
  - **if** $e \in T$ **then**
    - $C_e \leftarrow \{T[e]\}$
  - **else if** $G \wedge e \in M$ **then**
    - $C_e \leftarrow \{M[e]\}$
  - **else if** $G$ **then**
    - $c \leftarrow$ SIMCSERETRIEVE$(e, M)$
    - **if** $sim(c) > t_{e\text{-gold}}$ **then**
      - $C_e \leftarrow \{c\}$
  - **else**
    - $C_e \leftarrow$ FILTER(FACC1RETRIEVE$(e, k_e), score > t_e)$
- **return** $C$

`/* Pass 1: Entity Retrieval */`

**foreach** $b \in B$ **do**

- $C \leftarrow$ RETRIEVEENTITIES$(b, T, G, M, k_e, t_{e\text{-gold}}, t_e)$
- `/* Entity Permutation Evaluation */`
- $L_E \leftarrow$ GENERATEENTITYPERMUTATIONS$(b, C)$
- $L_E \leftarrow$ FIRSTK$(L_E, 500)$
- **foreach** $l \in L_E$ **do**
  - $q \leftarrow$ LOGICALFORMTOSPARQL$(l)$
  - $r \leftarrow$ EXECUTEQUERY$(q)$
  - **if** $r \neq \emptyset$ **then**
    - **return** $q, l, r$
  - **else**
    - $r \leftarrow$ EXECUTEQUERYWITHLABELFALLBACK$(q)$
    - **if** $r \neq \emptyset$ **then**
      - **return** $q, l, r$

`/* Pass 2: Entity and Predicate Retrieval */`

**foreach** $b \in B$ **do**

- $C \leftarrow$ RETRIEVEENTITIES$(b, T, G, M, k_e, t_{e\text{-gold}}, t_e)$
- $L_E \leftarrow$ GENERATEENTITYPERMUTATIONS$(b, C)$
- $L_E \leftarrow$ FIRSTK$(L_E, n_e)$
- **foreach** $l \in L_E$ **do**
  - `/* Predicate Retrieval */`
  - $N \leftarrow$ RETRIEVETWOHOPNEIGHBOURHOODOFENTITIES$(l)$
  - **foreach** *predicate mention* $p \in l$ **do**
    - $C_p \leftarrow \emptyset$
    - $C_p \leftarrow$ SIMCSERETRIEVE$(p, N)$
    - $C_p \leftarrow$ FILTER$(C_p, score > t_p)$
    - $C_p \leftarrow$ TOPK$(C_p, k_p)$
  - `/* Predicate Permutation Evaluation */`
  - $L_R \leftarrow$ GENERATEPREDICATEPERMUTATIONS$(l, C)$
  - $L_R \leftarrow$ FIRSTK$(L_R, n_p)$
  - **foreach** $l' \in L_R$ **do**
    - $q' \leftarrow$ LOGICALFORMTOSPARQL$(l')$
    - $r' \leftarrow$ EXECUTEQUERY$(q')$
    - **if** $r' \neq \emptyset$ **then**
      - **return** $q', l', r'$
    - **else**
      - $r' \leftarrow$ EXECUTEQUERYWITHLABELFALLBACK$(q')$
      - **if** $r' \neq \emptyset$ **then**
        - **return** $q', l', r'$

**return** $\emptyset$

The Entity and Predicate Retrieval procedure receives the parameters shown in Algorithm 1. Prediction Beams contain the generated logical forms, while the type (Section 3.2.4) and gold (Section 3.2.3) entity maps provide additional entity linking information. The thresholds $k_e$, $k_p$, $t_e$, and $t_p$ control the number and quality of retrieved Candidates. Lastly, $n_e$ limits the number of Entity Permutations passed to predicate retrieval, while $n_p$ determines the upper limit of Predicate Permutations tried.

**Successful permutation** A successful permutation is a permutation for which the generated SPARQL query executes against the knowledge base and returns a non-empty result set. The first successful permutation is selected as the winner.

**Entity Retrieval** At the start of each item, entity retrieval is performed to replace the Entity Mentions in the prediction beam with actual Freebase MIDs. Entity Retrieval receives a prediction beam $b$, the Global Type Map $T$, the Gold Entity Map and flag $M$ and $G$ and finally, the thresholds $k_e$, $t_{e-gold}$ and $t_e$. After extracting all entity mentions inside the prediction beam, a direct lookup in the Global Type Map $T$ is tried. If that fails and gold Entities $G$ are used, a direct Gold Entity Map $M$ lookup is attempted. This is then extended to a SimCSE similarity check with threshold $t_{e-gold}$ between the entity mention and the Gold Entity Map's contents upon failure. If all of the previous attempts fail, an index derived from FACC1 is queried for the entity mention. This results in a maximum of $k_e$ candidates whose SimCSE or FACC1 index scores exceed $t_e$. As a result, each entity mention now has its own Candidate Set. In the next step, the mentions in the prediction beam are substituted by up to 500 different permutations using the various candidate sets. This results in a list of entity permutations that is now sequentially tried for execution.

**Executability Check with label Fallback** The entity permutation is converted to SPARQL. No explicit predicate retrieval is required at this stage because the Freebase predicate identifiers contained in the prediction can be directly used to construct the corresponding SPARQL query. After conversion, the corresponding SPARQL query is executed against the knowledge base to get the result set. If the query delivers at least a single result, the permutation is considered successful. If, however, the query returns empty results, a second attempt is started. This time, the entity identifiers in the original query pattern are replaced by variables. The original identifiers are retained only to retrieve their labels, which are then used to constrain the replacement variable. For example:

```
PREFIX ns: http://rdf.freebase.com/ns/
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema

SELECT DISTINCT ?x
WHERE {
  FILTER (!isLiteral(?x) || lang(?x) = "" || langMatches(lang(?x), "en")
      )
  ns:m.05xyz12 ns:people.person.place_of_birth ?x .
}
```

**Code 8:** Example SPARQL query without label fallback

Suppose this query does not return any results. It is then transformed into the following alternative query:

```
PREFIX ns: http://rdf.freebase.com/ns/
PREFIX rdfs: http://www.w3.org/2000/01/rdf-schema

SELECT DISTINCT ?x
WHERE {
  FILTER (!isLiteral(?x) || lang(?x) = "" || langMatches(lang(?x), "en")
      )

  ns:m.05xyz12 rdfs:label ?en0 .
  ?ei0 rdfs:label ?en0 .
  FILTER(langMatches(lang(?en0), "EN"))

  ?ei0 ns:people.person.place_of_birth ?x .
}
```

**Code 9:** Example SPARQL query with label fallback

If this new query now delivers a non-empty result set, the entity permutation it originated from is marked successful. Otherwise, this entity permutation did not succeed, and the next one will be tried for executability next. If no entity permutation in any prediction beam was successful, the algorithm moves on to the second pass. While iterating over all prediction beams again, the same Entity Retrieval step is performed to get a list of entity permutations.

**Predicate Retrieval** For each entity permutation, all entity mentions are extracted and used to retrieve the predicate neighborhood. This is done through 4 separate SPARQL queries per entity mention. In the following diagrams, $e$ denotes the entity for which the predicate neighborhood is retrieved, $v1$ and $v2$ denote neighboring entities, and $p0$ and $p1$ denote predicates.

$$v_2 \xrightarrow{p_1} v_1 \xrightarrow{p_0} e \qquad v_2 \xleftarrow{p_1} v_1 \xrightarrow{p_0} e \qquad e \xrightarrow{p_0} v_1 \xleftarrow{p_1} v_2 \qquad e \xrightarrow{p_0} v_1 \xrightarrow{p_1} v_2$$

After building the predicate neighborhood pool, the SimCSE similarity between every Predicate Mention in the current entity permutation and the pool is calculated. The candidate set for every predicate mention consists of a maximum of $k_p$ predicates that exceed the score of $t_p$. Afterwards, the predicate mentions in the entity permutation are substituted by up to $n_p$ different combinations, again based on the different candidate sets. This results in a new list of predicate permutations, which are again tried for executability in the same way as entity permutations were. Processing every item in the dataset like this results in a final list of successful permutations and SPARQL queries. The corresponding result sets are finally compared to the gold answers to get an F1 score.

### 3.4.2 Extension

Due to the original algorithm being very rigid and explicit, the extended version aims to allow for greater control over the algorithm itself. The algorithm is now restricted to the core idea of ChatKBQA: Iterate through all prediction beams and do sequential trial for executability until a successful permutation is found. To make this work, a knowledge-base-specific module, as well as a list of entity and predicate linkers is used. These are defined after the algorithm itself in Section 4.1. In the context of Algorithm 2, bold symbols denote lists of pass-specific parameters, while their non-bold counterparts denote the value associated with the current pass.

**Algorithmus 2 :** Unsupervised Retrieval for Entities and Predicates (Extension)

**Input :** Prediction beams $B$, list of entity linkers $link_e$, list of predicate linkers $\mathbf{link}_p$, linker hyperparameters $link_{hp}$, knowledge-base specific module $kb$, number of entity permutations per pass $\mathbf{k}_e$, score threshold for entity permutations $\mathbf{t}_e$, number of predicate permutations per pass $\mathbf{k}_p$, score threshold for predicate permutations $\mathbf{t}_p$, beam limit per pass $\mathbf{k}_b$, optional entity-label fallback $f_{\text{label}}$

**Output :** successful SPARQL query $q^*$

**foreach** *predicate linker* $l_p$*, index* $idx$ *in* $\mathbf{link}_p$ **do**
  $B' \leftarrow$ LIMITBEAMS$(B, \mathbf{k}_b[idx])$;
  **foreach** *beam* $b \in B'$ **do**
    `/* Entity Retrieval Chain */`
    $EntMentions \leftarrow kb.$EXTRACTENTITIES$(b)$;
    $C_e \leftarrow \emptyset$;
    $unresolved \leftarrow EntMentions$;
    **foreach** *entity linker* $l_e \in \mathbf{link}_e$ **do**
      **if** $unresolved = \emptyset$ **then**
        **break**;
      $C_e^{new} \leftarrow l_e.$LINK$(unresolved, link_{hp})$;
      $C_e \leftarrow C_e \cup C_e^{new}$;
      $unresolved \leftarrow unresolved \setminus$ MENTIONS$(C_e^{new})$;
    $EntPerm \leftarrow$ PERMUTEBYENTITY$(C_e, \mathbf{k}_e[idx], \mathbf{t}_e[idx])$;
    **foreach** $EP \in EntPerm$ **do**
      `/* Predicate Retrieval */`
      $PredMentions \leftarrow kb.$EXTRACTPREDICATES$(EP)$;
      $C_p \leftarrow l_p.$LINK$(PredMentions, link_{hp}, EP)$;
      $PredPerm \leftarrow$ PERMUTEBYPREDICATE$(C_p, \mathbf{k}_p[idx], \mathbf{t}_p[idx])$;
      **foreach** $PP \in PredPerm$ **do**
        `/* Substitute retrieved entities and predicates */`
        $resolved \leftarrow$ KB.SUBSTITUTE$(b, EP, PP)$;
        $Q \leftarrow$ TOSPARQL$(resolved)$;
        `/* Executability Check */`
        **foreach** $q \in Q$ **do**
          $r \leftarrow$ EXECUTEQUERY$(q)$;
          **if** $r \neq \emptyset$ **then**
            **return** $q$;
        **if** $f_{\text{label}} =$ *true* **then**
          $q_f \leftarrow$ ENTITYLABELFALLBACK$(Q)$;
          **if** $q_f \neq \emptyset$ **then**
            $r \leftarrow$ EXECUTEQUERY$(q_f)$;
            **if** $r \neq \emptyset$ **then**
              **return** $q_f$;
**return** $\emptyset$;

The extended version receives a slightly different set of parameters. Instead of relying on fixed retrieval mechanisms, it receives lists of entity and predicate linkers, together with their corresponding hyperparameters. The knowledge-base-specific module provides the operations required to interpret and manipulate prediction beams for the target knowledge base. The remaining parameters control the number and quality of candidate permutations considered during retrieval, as well as optional execution behavior. A detailed explanation of every parameter can be found in Table 3.

| Parameter | Symbol | Description |
|---|---|---|
| Prediction beams | $B$ | Set of raw model predictions processed by the retrieval algorithm. |
| Entity linkers | $link_e$ | Ordered list of pre-defined entity linkers. The same set of linkers is used for every pass |
| Predicate linkers | $\mathbf{link}_p$ | Ordered list of a single pre-defined predicate linker per pass. |
| Linker hyperparameters | $link_{hp}$ | Configuration parameters passed to the entity and predicate linkers. |
| Knowledge-base-specific module | $kb$ | Provides knowledge-base-dependent operations, such as extracting entity and predicate mentions or substituting retrieved candidates into a prediction. |
| Entity permutation parameters | $\mathbf{k}_e, \mathbf{t}_e$ | $k_e$ specifies the maximum number of entity permutations considered per pass, while $t_e$ defines the minimum score required for an entity permutation to be considered during permutation generation. |
| Predicate permutation parameters | $\mathbf{k}_p, \mathbf{t}_p$ | $k_p$ limits the number of predicate permutations considered per pass, while $t_p$ specifies the minimum predicate permutation score. |
| Beam limit | $\mathbf{k}_b$ | Limits the number of prediction beams processed in each pass, allowing the computational cost of retrieval to be controlled independently of the total number of generated beams. |
| Entity-label fallback | $f_{\text{label}}$ | Controls whether the optional label-based fallback is applied when direct execution of a resolved query produces an empty result set. |

**Table 3:** Parameters of the extended entity and predicate retrieval procedure.

Each specified predicate linker represents a separate pass. For each predicate linker, the prediction beams are capped to the defined limit $k_b$ for this pass. The algorithm then

iterates through the prediction beams.

**Entity Retrieval** Every entity mention is extracted from the prediction beam using the knowledge-base-specific module. Afterwards, all specified entity linkers are called in sequence to get a scored candidate set for every entity mention. If a linker returns candidates for an entity mention, that mention is removed from the set of unresolved entity mentions. All unresolved mentions are then passed to the next entity linker. This allows for a linking cascade of increasing complexity and cost, while eliminating easy mentions early. If all entity mentions are successfully linked, entity retrieval exits early. After entity retrieval, we have a scored candidate set for every entity mention. Using this, the $k_e$ best-scoring entity permutations whose scores exceed $t_e$ are generated. In this case, $k_e$ and $t_e$ refer to the values associated with the current predicate linking pass.

**Predicate Retrieval** For each of these entity permutations, predicate retrieval is performed. This uses the predicate linker of the current pass. Every extracted predicate mention from the prediction beam along with the current entity permutation are passed to the predicate linker of the current pass. The predicate linker uses this information to return a scored candidate set for every predicate mention. After predicate retrieval, the scored candidate sets are used to generate up to $k_p$ predicate permutations whose scores exceed $t_p$. $k_p$ and $t_p$ again refer to the current predicate linker pass values.

**Executability Check** In the final step, the newly generated predicate permutations are iterated over. The current entity and predicate permutation are substituted into the original prediction beam and converted into a SPARQL query. Finally, an executability trial with optional label fallback decides whether the SPARQL query is successful by checking for empty results. This step is unchanged from the original.

## 3.5 Evaluation

A retrieval candidate is considered successful if the resulting SPARQL query can be executed against the knowledge base and returns a non-empty result. This criterion only establishes that a query is executable, however, and does not determine whether its result is correct. The retrieved queries therefore require a separate evaluation against the gold answers.

### 3.5.1 Original Approach

The original ChatKBQA evaluation uses Exact Match, F1, and Hit@1 to evaluate the answers produced by the retrieval algorithm. The precise evaluation differs slightly between CWQ and WebQSP due to the structure of their respective datasets.

**CWQ** For CWQ, predicted and gold answers are represented as sets. Exact Match is achieved if the predicted and gold answer sets are identical. Precision and recall are computed from the number of answers shared by the two sets, and F1 is their harmonic mean. Hit@1 is one if at least one predicted answer occurs in the gold answer set and zero otherwise. If both the predicted and gold answer sets are empty, all metrics are set to their perfect values. If only one of the two sets is empty, all metrics are set to zero. The reported metrics are obtained by averaging the per-question scores over the dataset.

**WebQSP** For WebQSP, a question can contain multiple annotated parses, each with its own gold answer set. Precision, recall, F1, and Hit@1 are therefore computed separately for each valid parse. The parse yielding the highest F1 is used for the reported precision, recall, and F1, while Hit@1 is one if at least one of the valid parses contains an answer that overlaps with the prediction. Empty-answer cases are handled separately by the WebQSP evaluation implementation. If both the predicted and gold answer lists are empty, all metrics are set to their perfect values. If the gold answer list is empty while the prediction is non-empty, F1 is zero, but the implementation assigns a recall of $1$ and Hit@1 of $1$. Conversely, if the gold answer list is non-empty and the prediction is empty, precision is set to $1$, while recall, F1, and Hit@1 are zero.

### 3.5.2 Extension

The extended evaluation uses Exact Match, Assignment F1, and Hit@1.

**Assignment F1** Assignment F1 follows the evaluation procedure also used by GRASP [26]. It computes the overlap between predicted and gold result rows by assigning predicted rows to gold rows and calculating the resulting precision, recall, and F1 score. This row-based evaluation is particularly relevant for knowledge bases such as Wikidata, where dataset queries may contain multiple `SELECT` variables. Such queries return a set of result rows, instead of a single list of individual answers. Assignment-F1 accounts for this structure by matching predicted rows to gold rows while penalizing missing columns. Additional columns not appearing in the gold rows are not penalized.

**Exact Match** Exact Match is defined as $1$ if the predicted and gold result rows are identical, independent of their order, and $0$ otherwise.

**Hit@1** Hit@1 is defined as $1$ if the resulting assignment-F1 is greater than zero and $0$ otherwise. As such, Hit@1 indicates whether at least one predicted result row overlaps with a gold result row.

The evaluation also performs several checks that affect which items contribute to the reported metrics. For each item, the selected SPARQL query is executed against the knowledge base and its result is compared with the gold result. The gold result can be obtained by executing the gold SPARQL query against the evaluation endpoint. For the experiments in this work, live gold evaluation is used. An item is marked as *stale* if its gold query does not produce any results. Stale items are excluded from the aggregate evaluation metrics rather than being counted as incorrect or correct. Predicted queries that cannot be executed, produce an error, or return an empty result are recorded separately and do not receive a positive score when a non-empty gold result is available. The evaluation additionally records the number of executable predictions and the execution status of each item. These checks allow retrieval failures and unavailable gold answers to be distinguished from actual answer mismatches in the reported results.

# 4 Implementation Details

This section describes the implementation of the entity and predicate linkers and the knowledge-base modules introduced in Section 3.4.2.

## 4.1 Entity and Predicate Linkers

Entity and predicate linkers define the retrieval strategies used to associate the entity and predicate mentions in a prediction with knowledge base identifiers. The extension provides a common interface for both types of linker. This allows different retrieval methods to be combined and exchanged without modifying the retrieval algorithm.

### 4.1.1 Definition

A linker receives a set of extracted labels together with contextual information about the current prediction and dataset item. The available inputs are summarized in Table 4.

| Input | Description |
|---|---|
| Labels | Entity or predicate labels extracted from the prediction beam that are to be linked. |
| Question | Original natural-language question associated with the current dataset item. |
| Prediction | Complete prediction beam from which the labels were extracted. |
| Item | Complete dataset item, providing access to item-specific information such as gold entity mappings. |
| Type map | Global type map generated during dataset preprocessing. It can be used by linkers that rely on previously extracted type information. |
| Entity map | Current entity permutation. This input is available to predicate linkers and allows them to consider the entities already retrieved when determining predicate candidates. |

**Table 4:** Inputs provided to entity and predicate linkers.

In addition to these inputs, each linker can receive a configuration of hyperparameters. This configuration is loaded once at the beginning of the retrieval procedure and provides linker-specific parameters without requiring the retrieval algorithm to depend on the implementation of a particular linker. A linker returns the outputs summarized in Table 5.

| Output | Description |
|---|---|
| Label map | Mapping from each successfully linked input label to its highest-scoring candidate. |
| Candidates | Mapping from each input label to a ranked list of candidate identifiers and their corresponding scores. |
| Failed labels | List of input labels for which the linker could not produce a candidate. |

**Table 5:** Outputs returned by entity and predicate linkers.

The candidate lists allow the retrieval algorithm to generate multiple permutations, while the label map provides the best-scoring candidate when only a single result is required. The failed-label list is used by the entity-linking cascade to determine which mentions must be passed to subsequent entity linkers. Both entity and predicate linkers follow this common interface. Predicate linkers additionally receive the current entity map, allowing them to take the entities already selected for the prediction into account when determining predicate candidates.

### 4.1.2 ChatKBQA Entity Linkers

To reproduce the original ChatKBQA entity-linking behavior within the extended retrieval framework, the original retrieval approach described in Section 3.4.1 is reimplemented as a sequence of individual linkers. The implementation follows the entity-linking logic of the original ChatKBQA approach. When executed sequentially, these linkers reproduce the original retrieval behavior while allowing individual retrieval strategies to be exchanged or extended independently.

**Type Map** The Type Map linker performs an exact lookup of each input entity label in the global type map (see Section 3.2.4). If a matching entry is found, it is returned as a candidate with a score of $1.0$. This corresponds to the first step of the original entity retrieval procedure. This linker exposes no parameters.

**Exact Gold** The Exact Gold linker performs an exact lookup of each input entity label in the gold entity map (see Section 3.2.3) associated with the current dataset item. A candidate is returned only if the entity mention in the prediction matches an entity mention contained in the gold mapping. As such, this linker only works for entities that appear in the Gold SPARQL query. Matching candidates are assigned a score of $1.0$. This linker represents the first half of handling the original retrieval's gold entity flag $G$. This linker exposes no parameters.

**SimCSE Gold** The SimCSE Gold linker extends the gold entity lookup to non-exact matches. It computes the SimCSE similarity between each input entity label and the entries of the gold entity map. Candidates whose similarity exceeds the configurable $gold_threshold$ are returned together with their corresponding similarity scores. This represents the second half of the original retrieval's gold entity behavior.

**FACC1** The FACC1 linker is the final and most general retrieval stage of the original entity retrieval procedure. It queries the FACC1 index using the input entity mentions and returns a ranked set of candidate entities. The linker has two hyperparameters: $top_k$, which limits the number of candidates considered from the FACC1 results, and $facc1_threshold$, which filters candidates according to their retrieval score. Candidates surviving both restrictions are returned together with their FACC1 scores. Compared to the preceding linkers, FACC1 retrieval is comparatively expensive and produces a broader set of candidates. It is therefore used as the final stage of the linker cascade, serving as a fallback for entity mentions that could not be resolved by the more specific retrieval strategies.

### 4.1.3 ChatKBQA Predicate Linkers

The original retrieval algorithm uses two different approaches for predicate retrieval. In the first pass, predicates are taken directly from the prediction. This is possible for Freebase because its predicate identifiers contain human-readable relation names. In the second pass, predicate retrieval is performed explicitly using the two-hop neighborhood of the entities contained in the current entity permutation. To replicate this behavior in the extended retrieval algorithm, we again implement their approach in the extension's linker framework.

**Simple** The Simple linker represents the first predicate retrieval pass of the original algorithm. It directly returns each input predicate label as its own candidate with a score of $1.0$. Consequently, no actual predicate retrieval is performed. During subsequent substitution, the original predicate label is therefore inserted back into the prediction unchanged. This represents the first pass of the original retrieval by just trusting the predicate mention in the beam to be correct. This linker exposes no parameters.

**Neighborhood** The Neighborhood linker represents the second predicate retrieval pass and corresponds to the neighborhood-based retrieval described in Section 3.4.1. For each entity in the current entity permutation, its two-hop predicate neighborhood is retrieved. The neighborhood consists of four possible traversal patterns. In the following diagrams, $e$ denotes the entity for which the predicate neighborhood is retrieved, $v_1$ and $v_2$ denote neighboring entities, and $p_0$ and $p_1$ denote predicates.

$$v_2 \xrightarrow{p_1} v_1 \xrightarrow{p_0} e \qquad v_2 \xleftarrow{p_1} v_1 \xrightarrow{p_0} e \qquad e \xrightarrow{p_0} v_1 \xleftarrow{p_1} v_2 \qquad e \xrightarrow{p_0} v_1 \xrightarrow{p_1} v_2$$

These four traversal patterns are implemented using four separate SPARQL queries for each entity. The resulting predicate pools are combined across all entities in the current entity permutation. Additionally, the raw predicate mentions are also seeded into the pool. For each input predicate label, SimCSE similarity is then computed against the predicates contained in the resulting pool. Candidates that satisfy the configured similarity threshold are retained, with at most $k$ candidates returned per input label. The linker exposes the parameters summarized in Table 6.

| Parameter | Description |
|---|---|
| `k` | Maximum number of candidates returned for each input label. |
| `t` | Minimum SimCSE similarity score required for a candidate to be retained. |
| `limit` | Maximum number of relations returned by each neighborhood query. |
| `timeout` | Maximum execution time permitted for each query against the knowledge base endpoint. |
| `model_name` | Name of the model used to compute the SimCSE similarity scores. |

**Table 6:** Parameters of the neighborhood predicate linker.

The implementation differs slightly from the original ChatKBQA procedure. The original implementation applies filtering conditions directly within the neighborhood queries and uses a query limit of $1000$. In the extended implementation, these filters are instead applied during post-processing, and the per-query limit is increased to $10000$. This can result in fewer than $1000$ candidates surviving the post-processing stage. This is because even though the query itself returns $10000$ candidates, almost all of them could in theory be filtered out in extreme cases. This would result in a pool of less than $1000$ surviving predicates. In practice, the larger initial pool provides more candidates for post-processing while avoiding the filtering conditions used by the original query formulation.

#### 4.1.4 Wikidata Entity Linkers

While the Type Map, Exact Gold and SimCSE Gold Linkers are knowledge base agnostic, the FACC1 Linker is Freebase-specific. As such, a new, broad entity linker for Wikidata is needed.

**WBSearchEntities** The WBSearchEntities linker uses Wikidata's `wbsearchentities` API to retrieve an initial set of entity candidates for each input entity mention. The API returns a ranked list of candidates based on its own search mechanism. However, selective inspection of failure cases showed that the returned ranking can place semantically unrelated entities with matching names above the more common entity. For example, searches for common city or country names may return albums, songs, or sports clubs containing the same name before the corresponding geographical entity. To improve the ranking of the retrieved candidates, a lightweight re-ranking pass is applied. For each candidate, three scoring components are combined: a semantic similarity score, a lexical matching score, and a score derived from the candidate's position in the API result list. The semantic score is computed using SimCSE. The query representation consists of the original question and the input entity mention, while each candidate is represented by its Wikidata label and description. The cosine similarity between these representations provides the semantic component of the score. The lexical component provides explicit bonuses for close matches between the input mention and the candidate label. An exact label match receives a bonus of $0.25$, while a substring match, where either string contains the other, receives a bonus of $0.12$. A whitespace-normalized label match receives a bonus of $0.15$. These bonuses provide additional preference for candidates whose labels closely correspond to the input mention. The third component makes use of the ranking returned by the Wikidata API. A rank score of $1/(i+1)$ is assigned to the candidate at position $i$ in the returned list. This preserves a preference for candidates appearing earlier in the API results. The three components are combined using the following weighted score:

$$s(c) = 0.55\, s_{\text{semantic}}(c) + 0.30\, s_{\text{lexical}}(c) + 0.15\, s_{\text{rank}}(c).$$

The resulting scores are used to rank the candidates. Candidates with a score below the configurable threshold `te` are discarded, and at most `ke` candidates are retained for each input mention. If all candidates fall below the threshold, the implementation retains the unfiltered candidate set as a fallback. The weights and lexical bonus values were chosen through limited manual experimentation on observed failure cases rather than systematic hyperparameter search. They should be considered as empirically selected configuration values rather than optimized parameters. The linker exposes the parameters summarized in Table 7.

| Parameter | Description |
|---|---|
| `ke` | Maximum number of candidates retained for each input entity mention. |
| `te` | Minimum hybrid score required for a candidate to be retained. |
| `search_limit` | Maximum number of candidates requested from the Wikidata `wbsearchentities` API for each input entity mention. |
| `semantic_weight` | Weight assigned to the SimCSE semantic similarity score. |
| `lexical_weight` | Weight assigned to the lexical matching score. |
| `rank_weight` | Weight assigned to the candidate's position in the API result list. |
| `model_name` | Name of the Sentence Transformer model used to compute the SimCSE similarity scores. |

**Table 7:** Parameters of the WBSearchEntities linker.

#### 4.1.5 Wikidata Predicate Linkers

Similarly to the FACC1 Linker, the Freebase linkers Simple and Neighborhood do not work out of the box for Wikidata. For this reason, we implement two linkers that replicate the original intent of the Freebase equivalents.

**Label Search** While Freebase predicate identifiers are already in natural language, this is not the case for Wikidata. In order to recreate the behavior of the Freebase Simple Linker, which mostly just trusts the prediction beam to be correct, this linker performs a label search against the knowledge base, restricted to properties that appear as `wikibase:directClaim` predicates. Candidates are first filtered by substring containment against the English label, then ranked into one of three confidence tiers depending on how close that match is: $1.0$ for an exact label match, $0.67$ for matches where the label begins with the predicate mention, and $0.33$ for matches that merely contain the predicate mention elsewhere. Within a tier, candidates are further ordered by label length, preferring shorter labels as a proxy for closeness to an exact match. In contrast with the Freebase variant, this linker returns a list of candidates.

**Neighborhood** This linker mirrors the structure of the Freebase variant Neighborhood, using the same four two-hop traversal patterns, but differs from it in two respects. First, Wikidata predicate identifiers (PIDs) carry no lexical content, unlike Freebase's human-readable predicate names, so candidate PIDs collected from the neighborhood traversal must first be resolved to their English `rdfs:label` in order to compute SimCSE similarity. Second, the Freebase variant's fallback step of also seeding the candidate pool with the raw extracted predicate labels is dropped here, since an unresolved label is not a

valid PID and cannot be substituted directly. The linker exposes the same parameters as shown in Table 6.

## 4.2 Knowledge Base Module

In order to keep the base retrieval algorithm (as described in Section 3.4.2) independent of knowledge-base-specific implementation details, these details are isolated in a dedicated KB module. The module is responsible for handling functionality that is different between knowledge bases. This allows the retrieval algorithm to operate against a common interface, regardless of which knowledge base is being used. Each KB module provides the functionality summarized in Table 8.

| Functionality | Description |
|---|---|
| Prefixes | Defines the prefixes and corresponding namespaces used by the knowledge base. |
| Label query | Defines the SPARQL query used to retrieve labels for entities and predicates. |
| Type query | Optionally defines a SPARQL query for determining which candidate URIs represent types or classes. This is used to populate the global type map (see Section 3.2.4). |
| URI normalization | Returns the label-bearing URI corresponding to a given URI. |
| Label parsing and formatting | Parses label-query results and converts URIs and labels into the representation used by the intermediate query representation. |
| Prediction extraction | Extracts entity and predicate mentions from a model prediction according to the syntax used by the knowledge base. |
| Candidate substitution | Replaces linked entity and predicate mentions in a prediction with their corresponding knowledge-base URIs. |
| Answer normalization | Converts returned URIs into knowledge-base-local identifiers for evaluation. |

**Table 8:** Knowledge-base-specific functionality provided by the KB module.

The common interface is implemented as an abstract base class, `BaseKB`. It provides functionality that is shared across knowledge bases, such as extraction of URIs from the intermediate representation, answer-URI normalization, and handling of common

SPARQL prefixes. Knowledge-base-specific behavior is defined through abstract methods and configuration attributes implemented by concrete subclasses. The extension provides separate Knowledge Base Modules for Freebase and Wikidata. Although both implementations expose the same interface, several operations differ because the two knowledge bases use different URI schemes and query representations.

**Freebase** The Freebase module defines the namespaces used for entities and predicates and the regular expressions required to identify them in predictions and intermediate representations. Freebase entities are represented by machine identifiers (MIDs), while predicates use hierarchical identifiers such as `domain.type.property`. Entity labels are retrieved using the Freebase `type.object.name` property. Since Freebase predicates do not provide a separate human-readable label in the same way as Wikidata properties, their displayed predicate representation is derived directly from the URI. The module also implements the extraction and substitution rules required by the Freebase prediction syntax. Entity mentions using the `fb:` prefix are replaced by their resolved MIDs, while predicate mentions using the `fbp:` prefix are replaced by their corresponding Freebase relation URIs. For Freebase, a URI is considered a type if it occurs as the object of the Freebase `type.object.type` relation with `type.type` as the object.

**Wikidata** The Wikidata module adapts the same interface to Wikidata's URI and predicate structure. Entities are represented by QIDs, while properties are represented by PIDs and may occur under several Wikidata property namespaces, such as `wdt:`, `p:`, and `ps:`. The module therefore handles these prefixes explicitly when extracting and substituting predicate mentions. Unlike Freebase, Wikidata properties have human-readable labels stored on their corresponding property entities. The module therefore normalizes property URIs to their Wikidata entity URIs when retrieving labels and converts the resulting labels back into the appropriate predicate representation. English labels are preferred, with a language-neutral fallback when available. The Wikidata module additionally defines the type query based on the `P31` (*instance of*) and `P279` (*subclass of*) properties. This is done because Wikidata does not use a single type relation analogous to Freebase's `type.object.type`. As such, the module interprets an entity as a type, if it is the object of one of the two Wikidata properties named above.

By handling these differences in the KB module, the retrieval algorithm can perform entity and predicate extraction, candidate substitution, label handling, and answer normalization without containing logic specific to any single Knowledge Base. Adding support for another knowledge base therefore primarily requires implementing the corresponding KB module rather than modifying the retrieval algorithm itself.

## 4.3 Complexity Analysis

The computational complexity of the retrieval procedure is primarily determined by the number of candidate SPARQL queries generated and executed for each dataset item. The internal complexity of individual entity and predicate linkers is not included in this analysis, as the linkers are configurable and may perform substantially different operations depending on the selected knowledge base and retrieval strategy. Their individual computational costs are discussed together with the corresponding linker implementations in Section 4.1. In this section, bold symbols denote lists of pass-specific parameters, while their non-bold counterparts denote the value associated with the current pass. The variables used to analyze the number of SPARQL candidates are summarized in Table 9.

| Variable | Description |
|---|---|
| $p$ | Number of predicate-linking passes. |
| $\mathbf{k}_b$ | Maximum number of prediction beams considered per pass. |
| $\mathbf{k}_e$ | Maximum number of entity permutations considered per beam. |
| $\mathbf{k}_p$ | Maximum number of predicate permutations considered per entity permutation. |
| $f_{label}$ | $1$ if the label fallback is used, $0$ otherwise. |

**Table 9:** Variables used in the complexity analysis.

For each predicate-linking pass $i$, at most $k_b$ prediction beams are considered. For each beam, the entity-linking stage generates at most $k_e$ entity permutations. Each entity permutation then produces at most $k_p$ predicate permutations. Each predicate permutation results in at most 1 final SPARQL candidates without label fallback, or 2 if label fallback is used. As such, with $p$ predicate-linking passes, the maximum number of SPARQL candidates considered for a dataset item is

$$N_{\text{query}}^{\max} = (1 + f_{label}) \sum_{i=0}^{p-1} k_b k_e k_p.$$

The complexity of the individual linkers depends on their implementation and configuration. For example, some linkers perform local lookups, whereas others execute queries against the knowledge base or external APIs and may additionally perform embedding-based similarity calculations. Since the set and order of linkers can be configured independently for each run, these costs are treated separately from the

common retrieval complexity analyzed here. The actual number of executed queries can be substantially lower than this upper bound. The retrieval procedure terminates as soon as a candidate produces a successful query result. Thus, the expression above represents the maximum number of SPARQL candidates considered rather than the typical number of queries executed for a successful retrieval.

# 5 Results

## 5.1 Hardware and Software Configuration

All experiments were performed on the university computing cluster. Jobs were submitted using Slurm and executed on the cluster's compute nodes. The resources allocated to the experiments are summarized in Table 10.

| Resource | Allocation |
|---|---|
| CPU | 1 CPU core per job |
| Memory | 130–200 GB for Freebase; 32 GB for Wikidata |
| GPU | NVIDIA L40S for SimCSE-based experiments, fine-tuning and inference |
| Cluster | University computing cluster |
| Knowledge base | Locally hosted Virtuoso for Freebase; externally hosted QLever instance for Wikidata |

**Table 10:** Computational resources allocated to the retrieval experiments.

Memory allocation depended on the knowledge base. The Freebase experiments used approximately 130–200 GB of memory due to the locally hosted Virtuoso instance and loading of the FACC1 index, whereas the Wikidata experiments used approximately 32 GB because the QLever endpoint was hosted on a separate machine. Virtuoso was used for Freebase to maintain comparability with the original ChatKBQA results, which were also obtained using the same knowledge base setup.

## 5.2 Validation of the Extended Implementation

Before evaluating the extensions introduced in this work, the original ChatKBQA retrieval procedure was reproduced. This serves to assess whether the extended implementation produces results comparable to the original system. The validation was performed on the CWQ [20] and WebQSP [31] test sets. The predictions released with the original ChatKBQA implementation were post-processed to work for the extension and the extension's Freebase knowledge base module. Code 10 shows an example of how the original predictions were transformed.

```
{
  "ID": "WebQTest-0",
  "debug_old_pred": [
    "( JOIN ( R [ location , country , languages spoken ] ) [ Jamaica ] )",
    "( JOIN ( R [ location , country , language ] ) [ Jamaica ] )"
  ],
  "predict": [
    "( JOIN (R fbp:location.country.languages_spoken) fb:Jamaica )",
    "( JOIN (R fbp:location.country.language) fb:Jamaica )"
  ]
}
```

**Code 10:** Example of adapting an original ChatKBQA prediction to the extended Freebase representation.

The original predictions use the label-based representation expected by the original retrieval implementation. They were converted to the prefixed Freebase representation used by the extended algorithm. The conversion only changes the syntactic representation of entities and predicates. The resulting prediction beams were afterwards processed by the extended retrieval algorithm and finally evaluated. The resulting metrics were then compared with the results reported in the original ChatKBQA publication. Table 11 shows the results of this comparison. The config used to produce each result in the extended pipeline can be viewed by clicking the run name. In this case, we match the original ChatKBQA parameters, which includes the label fallback.

| Run | EM | F1 | Hit@1 | Note |
|---|---|---|---|---|
| ChatKBQA CWQ | 73.3 | 77.8 | 82.7 | Official metrics reported in paper |
| Reproduction CWQ | 74.3 | 78.3 | 82.7 | Adapted original predictions |
| ChatKBQA CWQ Gold | 76.8 | 81.3 | 86.0 | Official metrics reported in the paper, using gold entities |
| Reproduction CWQ Gold | 78.3 | 82.2 | 86.3 | Adapted original predictions, using gold entities |
| ChatKBQA WebQSP | 73.8 | 79.8 | 83.2 | Official metrics reported in paper |
| Reproduction WebQSP | 73.3 | 77.9 | 82.4 | Adapted original predictions |
| ChatKBQA WebQSP Gold | 77.8 | 83.5 | 86.4 | Official metrics reported in paper, using gold entities |
| Reproduction WebQSP Gold | 76.7 | 81.1 | 85.3 | Adapted original predictions, using gold entities |

**Table 11:** Comparison of the reproduced ChatKBQA results with the reported results.

The reproduced metrics are close to the results reported by the original ChatKBQA publication, with differences of at most 1.5 percentage points for CWQ and 2.4 percentage points for WebQSP. However, the discrepancy in F1 score on WebQSP can not solely be attributed to retrieval performance, as the evaluation implementation for WebQSP differs between the extension and original (see Section 3.5). While the original compares the predicted answer set to the gold answer set of all parses of an item and finally takes the highest-scoring one among them, the extension only compares the answer set to the first parse that was successfully converted to the training format. In the extension, nearly all gold answer sets are as such the first parse of each original dataset item. This provides evidence that the extended pipeline's results are reasonably comparable with the original ChatKBQA predictions and establishes a baseline against which the subsequent experiments can be evaluated.

## 5.3 Retraining of the Original Models

After validating the extended retrieval and evaluation procedure, the models used to generate the predictions were retrained using the training procedure implemented in this work. This allows the effect of retraining to be evaluated separately from the changes to the retrieval and evaluation pipeline. The retrained Llama 2 [21] models were evaluated using the same retrieval configurations as the reproduced ChatKBQA runs shown in Table 11. This keeps the entity and predicate linking procedures, beam sizes, and retrieval parameters fixed, allowing the results to be compared directly with the adapted

predictions used in the previous experiment. Table 12 compares the results obtained from the adapted original predictions with those obtained using the retrained Llama 2 models. We use the same base models as ChatKBQA: Llama-2-7B for WebQSP and Llama-2-13B for CWQ. Additionally, the training config is mirrored as closely as possible to the original train command used by ChatKBQA.

Note that from this point onwards, experiments do not use the label fallback. The distinct impact of the label fallback on the metrics is shown in Appendix A.1. Due to this, the reproduced metrics of the original ChatKBQA predictions are lower in this table compared to Table 11.

| Run | EM | F1 | Hit@1 | time | Note |
|---|---|---|---|---|---|
| Reproduction CWQ | 71.7 | 75.8 | 80.0 | 15.75s | Adapted original predictions |
| Llama-2-13b_CWQ-sparql | 72.8 | 76.5 | 80.5 | 18.28s | SPARQL retrain |
| Reproduction CWQ Gold | 78.3 | 82.2 | 86.3 | 4.59s | Adapted original predictions, using gold entities |
| Llama-2-13b_CWQ-sparql Gold | 80.5 | 83.8 | 87.5 | 4.33s | SPARQL retrain, using gold entities |
| Reproduction WebQSP | 73.0 | 77.5 | 82.0 | 18.00s | Adapted original predictions |
| Llama-2-7b_WebQSP-sparql | 72.0 | 76.3 | 80.5 | 15.56s | SPARQL retrain |
| Reproduction WebQSP Gold | 76.5 | 81.0 | 85.3 | 14.19s | Adapted original predictions, using gold entities |
| Llama-2-7b_WebQSP-sparql Gold | 77.0 | 81.0 | 85.3 | 5.95s | SPARQL retrain, using gold entities |

**Table 12:** Comparison of adapted prediction results with the retrain results.

The reproduced metrics are comparable to the values produced using the adapted original predictions. The maximum F1 score discrepancy is 1.6 percentage points for CWQ and 1.2 percentage points for WebQSP.

## 5.4 Hyperparameter Analysis

This section analyzes the hyperparameter configuration used in the original ChatKBQA experiments.

### 5.4.1 ChatKBQA Hyperparameter Configuration

Table 13 shows the hyperparameter configuration used for the retraining experiments. The parameters were adopted from the original ChatKBQA entity and predicate retrieval implementation without modification. The experiments are the same as in Table 12. As such, the run 'CWQ', for example, refers to 'Llama-2-13B CWQ' in that table.

| | | Simple | | Neighborhood | |
|---|---|---|---|---|---|
| **Run** | $\mathbf{k}_b$ | $k_e$ | $k_p$ | $k_e$ | $k_p$ |
| CWQ | 8 | 500 | 1 | 30 | 300 |
| WebQSP | 15 | 500 | 1 | 50 | 4000 |

**Table 13:** Hyperparameter setup used by the retraining experiments, taken from original ChatKBQA retrieval code [15].

Using the complexity analysis from Section 4.3, the maximum amount of SPARQL queries candidates tried for execution against the endpoint can be calculated.

**CWQ** For CWQ, the maximum number of prediction beams is 8. For all of these, a maximum of 500 entity permutations are generated by the Simple Linker's entity retrieval. Each of these entity permutations then results in a single predicate permutation and as such, a single SPARQL query candidate. The maximum number of SPARQL query candidates resulting from this pass are

$$N_{query}^{simple} = 8 \cdot 500 \cdot 1 = 4000$$

Assuming all of these queries retrieve empty results, the algorithm proceeds to the Neighborhood Linker pass. During this pass, the 8 prediction beams each produce 30 entity permutations. All of these entity permutations are expanded into 300 predicate permutations. With this, the maximum number of SPARQL query candidates for this pass is

$$N_{query}^{neighborhood} = 8 \cdot 30 \cdot 300 = 72,000$$

The maximum number of SPARQL query candidates is given by

$$N_{query}^{total} = N_{query}^{simple} + N_{query}^{neighborhood} = 76,000$$

The maximum number of SPARQL query candidates that can be executed for a single item in CWQ is $76.000$. Note that this number does not include any additional queries that are fired against the endpoint by the Neighborhood itself.

**WebQSP** In the case of WebQSP, there can be a maximum of 15 prediction beams. For each prediction beam, the first entity retrieval is performed and 500 distinct entity permutations are generated. Each of these entity permutations go through the first predicate linking step, which results in 1 predicate permutation. As such, the maximum amount of SPARQL query candidates generated in the Simple Linker pass is

$$N_{query}^{simple} = 15 \cdot 500 \cdot 1 = 7500$$

Assuming none of these candidates successfully retrieve a non-empty answer, the Neighborhood pass starts. For all of the 15 prediction beams, 50 entity permutations are generated. Once more, each entity permutation results in 4000 predicate permutations, which leads to 4000 SPARQL query candidates. With this, the maximum amount of SPARQL query candidates generated in this pass is

$$N_{query}^{neighborhood} = 15 \cdot 50 \cdot 4000 = 3,000,000$$

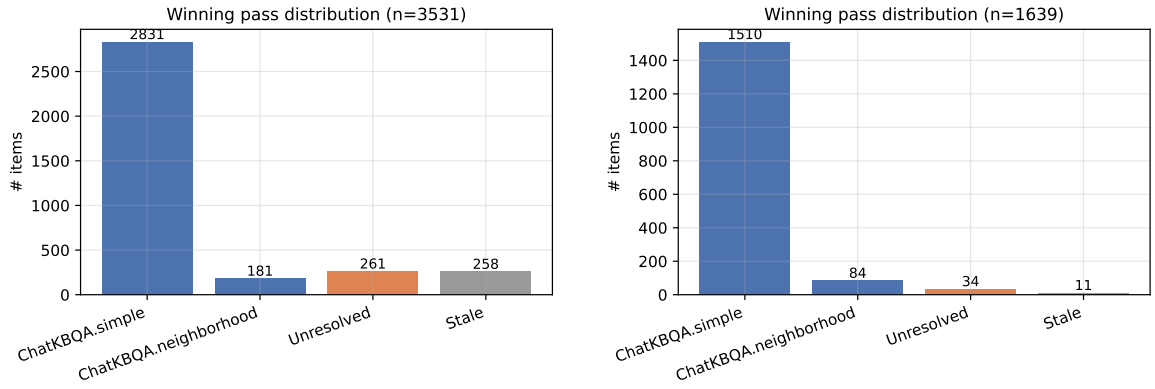
The maximum number of SPARQL query candidates is given by

$$N_{query}^{total} = N_{query}^{simple} + N_{query}^{neighborhood} = 3,007,500$$

This results in a maximum of $3,007,500$ SPARQL query candidates being executed against the endpoint. Like CWQ, this only includes actual candidate SPARQL queries and does not include any linking-related query executions.

### 5.4.2 Runtime and Performance per Retrieval Pass

Figure 5 shows which pass resolved how many items for the displayed experiments. Dataset items where the gold SPARQL query returns empty results upon execution are treated as stale and are collected separately.




**(a)** CWQ **(b)** WebQSP

**Figure 5:** Distribution of items resolved by the different retrieval passes on retrain experiments.

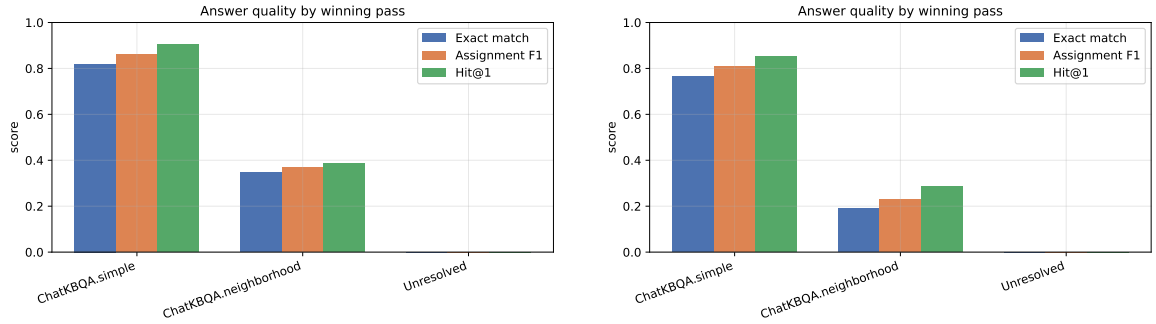
Evidently, the majority of items across all experiments are resolved by the Simple Linker pass, while only a small portion of items are resolved by the Neighborhood Linker. Another small portion of items are not resolved at all.

To establish a baseline for runtime per pass, Table 14 shows the runtime per predicate linker pass for the retrained experiments shown in Figure 5. In the table, '#' refers to the number of items resolved by the pass, while 'Time' refers to the average runtime in seconds over all items of the specified pass. Experiments can define a timeout threshold. If a dataset item is neither resolved nor finished within the specified threshold, the item is treated as unresolved. This is used for experiments where certain items otherwise take multiple hours to resolve. Furthermore, items deemed as stale are instantly skipped by the retrieval step and as such do not demand any runtime.

| | Total | Simple | | Neighborhood | | Unresolved | | Timeout | |
|---|---|---|---|---|---|---|---|---|---|
| **Run** | **Time** | **#** | **Time** | **#** | **Time** | **#** | **Time** | **#** | **Time** |
| CWQ | 18.28 s | 2882 | 0.18 s | 196 | 69.01 s | 453 | 111.46 s | – | – |
| WebQSP | 15.56 s | 1510 | 0.06 s | 84 | 95.12 s | 26 | 300.18 s | 8 | 1200,67 s |

**Table 14:** Distribution of items resolved and average retrieval runtime by predicate linking pass.

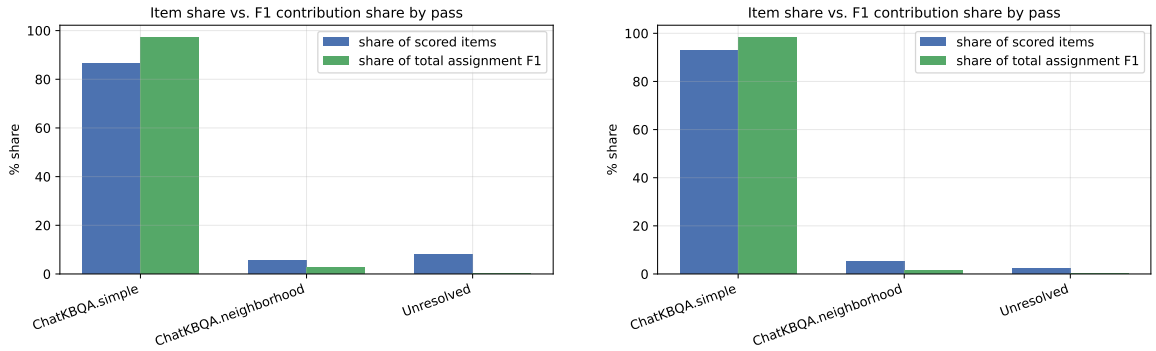
As shown in the table, the Simple Linker has a quick runtime to find an executable result. The Neighborhood Linker has a substantially longer runtime to find a result. Unresolved items have the longest average runtime, as those items go through both passes before ultimately failing to find an executable query. Another important consideration is, that even though a certain linker might resolve many items, a resolved item is just a SPARQL query that has non-empty results when executing against the endpoint. Whether these results match the actual gold answer of the dataset item is not guaranteed. Figure 7 shows the average metrics per predicate linker pass.




**(a)** CWQ **(b)** WebQSP

**Figure 7:** Exact Match, Assignment F1 and Hit@1 per predicate linker pass on retrain experiments.

While items resolved by the Simple Linker have an average Assignment F1 score of over 80% in all experiments, the performance of the Neighborhood Linker has a much bigger variance. Items resolved by it only score an average Assignment F1 score of 36.8% for CWQ and 23.0% for WebQSP . To put this into perspective, Figure 9 instead shows the share of items resolved and the share of total Assignment F1 score contribution by each predicate linker pass for the same experiments.




**(a)** CWQ **(b)** WebQSP

**Figure 9:** Share of resolved items and total Assignment F1 score contribution per predicate linker pass on retrain experiments.

For CWQ , the Simple Linker resolves 86.5% of scored items and contributes approximately 97.4% of the total assignment F1, while the Neighborhood Linker resolves 5.5% of the items and contributes approximately 2.6% to the total Assignment F1 score. For WebQSP , the Simple Linker resolves 92.8% of scored items and contributes approximately 98.4% of the total assignment F1, while the Neighborhood Linker resolves 5.2% of the items and contributes approximately just 1.6% to the total Assignment F1 score.

### 5.4.3 Winning Query Candidate Distributions

The results up to this point indicate that the Neighborhood pass contributes relatively little to the overall retrieval performance while requiring substantially more computation per item. For a given dataset item, the winning index denotes the position within a permutation set at which the item is resolved. This is recorded separately for the prediction beam, the entity permutation, and the predicate permutation. For example, a beam index of $4$ indicates that the fourth prediction beam produced the candidate that ultimately resolved the item, while an entity permutation index of $11$ indicates that the eleventh entity permutation of that beam led to the resolving candidate. The same applies to the predicate permutation index. For the purpose of visual clarity, these values are summarized in Table 15. The *Cap* column denotes the actual hyperparameter config value used by the specific experiment. Appendix A.2 instead shows the full distribution of winning candidate indexes for each predicate linker pass across the established set of experiments. In the table, *EP* denotes the entity permutation index and *PP* denotes the predicate permutation index. The P90, P95, and P99 values denote the 90th, 95th, and 99th percentiles of the winning candidate index, respectively. For example, P90 indicates the index value at or below which 90% of resolved items are located.

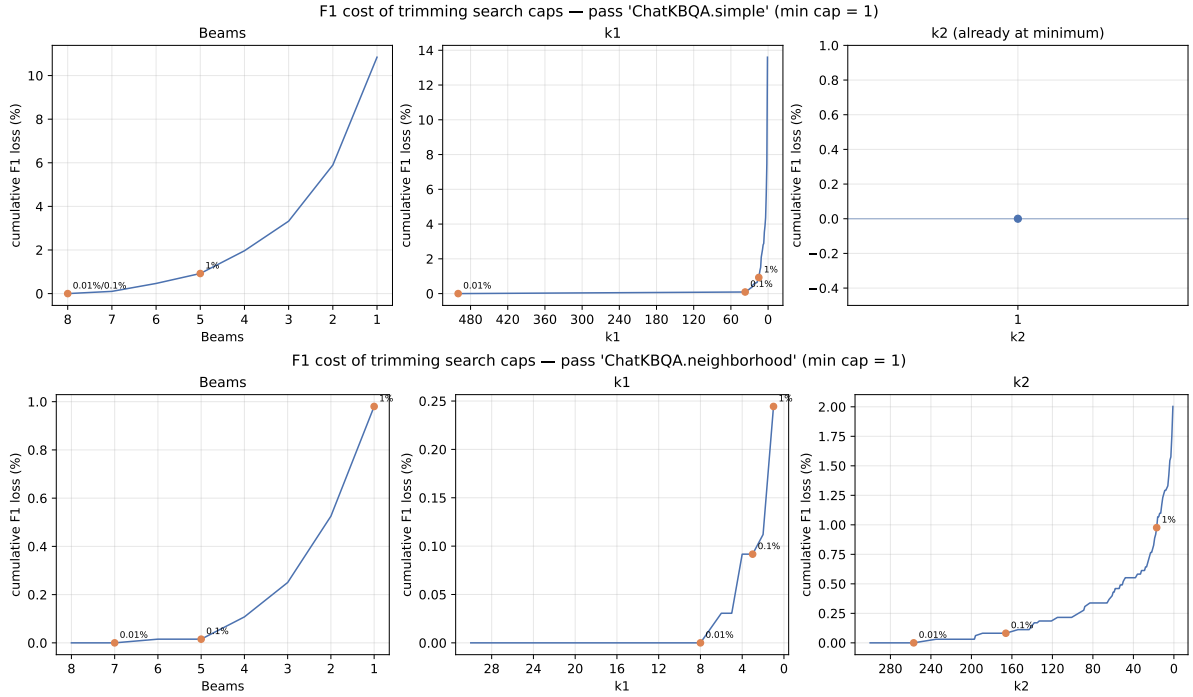
| Run | Index | Simple | | | | | | Neighborhood | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Cap | Mean | Max | P90 | P95 | P99 | Cap | Mean | Max | P90 | P95 | P99 |
| CWQ | Beam | 8 | 1.4 | 8 | 2.0 | 4.0 | 6.0 | 8 | 2.2 | 8 | 5.0 | 6.0 | 7.0 |
| | EP | 500 | 1.9 | 38 | 2.0 | 6.0 | 17.0 | 30 | 2.0 | 28 | 4.0 | 6.0 | 15.6 |
| | PP | 1 | 1.0 | 1 | 1.0 | 1.0 | 1.0 | 300 | 55.5 | 296 | 144.0 | 187.0 | 240.4 |
| WebQSP | Beam | 15 | 1.5 | 14 | 3.0 | 5.0 | 9.9 | 15 | 2.5 | 12 | 6.7 | 8.0 | 12.0 |
| | EP | 500 | 1.5 | 76 | 1.0 | 3.0 | 12.0 | 50 | 1.7 | 50 | 1.0 | 2.0 | 11.8 |
| | PP | 1 | 1.0 | 1 | 1.0 | 1.0 | 1.0 | 4000 | 107.4 | 2997 | 96.4 | 229.3 | 2027.6 |

**Table 15:** Summary statistics of the winning candidate indices for each retrain experiment and predicate linker pass. EP denotes the entity permutation index, PP the predicate permutation index, and Beam the beam rank.

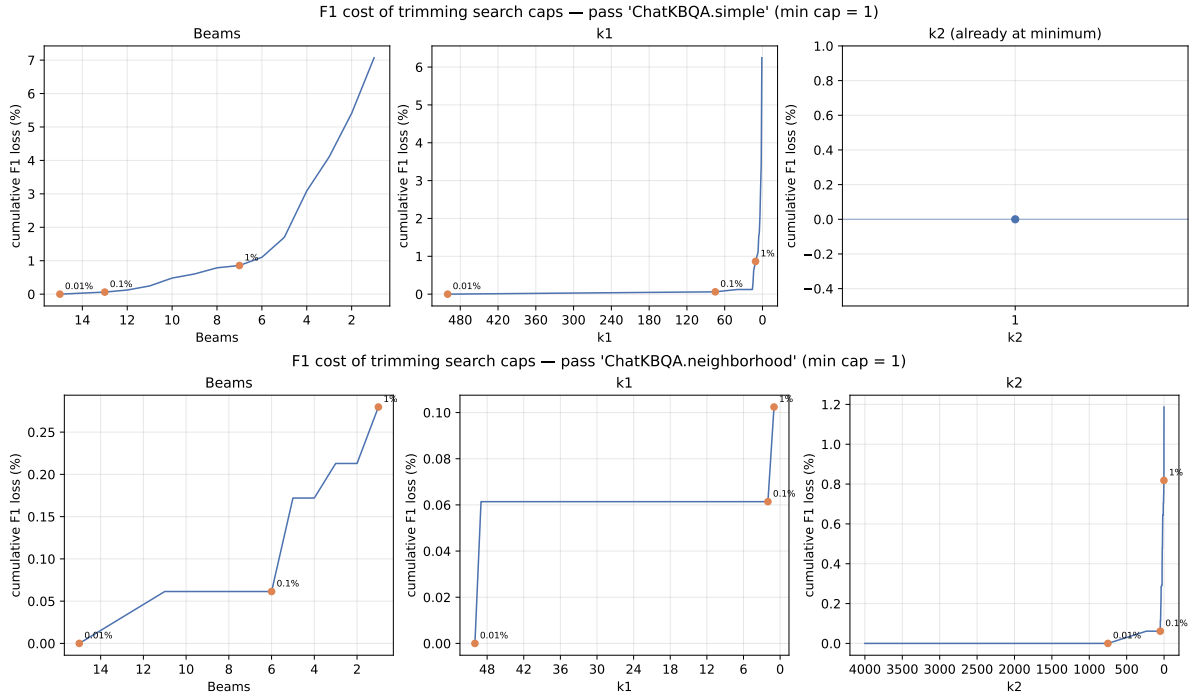
The table shows that the configured retrieval caps are generally substantially higher than the observed winning candidate indices. This is particularly pronounced for the entity permutation parameter. For example, the Simple Linker uses an entity permutation cap of 500 for both CWQ and WebQSP, while the P99 of the winning entity permutation index is only 17 and 12, respectively. Similarly, the Neighborhood Linker uses caps of 30 and 50, while the corresponding P99 values are 15.6 and 11.8. As such, only a small fraction of resolved items require candidate positions close to the configured upper bounds. The same pattern can be observed for the beam parameter, although the difference between the configured cap and the observed winning indices is less pronounced.

### 5.4.4 Hyperparameter sensitivity

The previous analysis showed that the configured retrieval caps are substantially higher than the candidate indices required by most resolved items. This raises the question of how much these parameters can be reduced without significantly affecting retrieval performance. To investigate this, each retrieval parameter was reduced incrementally while keeping the remaining parameters fixed at their original values. The resulting change in Assignment F1 was recorded to quantify the sensitivity of retrieval performance to each parameter. Figure 10 shows the cumulative Assignment F1 loss as a function of the respective parameter value. The loss is measured relative to the original configuration, such that a value of zero indicates no loss compared to the full retrieval configuration. This allows the point at which reducing a parameter begins to noticeably affect retrieval performance to be identified. Note that the cumulative Assignment F1 loss at any given point is an upper bound. The real loss could be less significant, because a later beam, entity permutation or even predicate linker pass could still resolve the item.




**(a)** CWQ




**(b)** WebQSP

**Figure 10:** Hyperparameter sensitivity per predicate linker pass for retrain experiments.

The resulting curves show that retrieval performance is relatively insensitive to substantial reductions of several retrieval parameters. In particular, the entity and predicate permutation limits can be reduced considerably before a noticeable loss in Assignment F1 occurs. The number of prediction beams is generally the most sensitive to reduction.

### 5.4.5 Retrieval Parameter Reduction

Based on the insights of Figure 10, we propose a new hyperparameter configuration. The goal of this configuration is to provide a comparable and faster set of hyperparameters for the remaining experiments. Table 16 shows this reduced configuration.

| | Pass 1 | | Pass 2 | |
|---|---|---|---|---|
| $\mathbf{k}_b$ | $k_e$ | $k_p$ | $k_e$ | $k_p$ |
| 8 | 100 | 5 | 5 | 15 |

**Table 16:** Reduced retrieval hyperparameter configuration.

Additionally, Table 17 defines the exact linker configuration used for further experiments.

| Knowledge Base | Entity Linkers | Predicate Linkers |
|---|---|---|
| Freebase | Type Map<br>FACC1 | Simple<br>Neighborhood |
| Freebase (Gold) | Type Map<br>Exact Gold<br>SimCSE Gold<br>FACC1 | Simple<br>Neighborhood |
| Wikidata | Type Map<br>WBSearchEntities | Label Search<br>Neighborhood |
| Wikidata (Gold) | Type Map<br>Exact Gold<br>SimCSE Gold<br>WBSearchEntities | Label Search<br>Neighborhood |

**Table 17:** Entity and predicate linker configurations used for the different knowledge bases and gold settings.

The reduced configuration is not the result of a formal hyperparameter optimization. Instead, the values were selected heuristically based on the observed winning index distributions and sensitivity analysis. The selected values provide substantial reductions in the retrieval search space while retaining a margin beyond the candidate positions required by most resolved items. The configuration is intended as a practical reduction of the original search space rather than as an optimized parameter setting.

**Training Hyperparameters** Further experiments use Qwen2.5-7B [18] instead of the Llama 2 family [21]. We use LoRA (see Section 2.8), with $r = \alpha = 32$, targeting all linear layers. Training is done for 4 epochs with a learning rate of $1e^{-4}$. The training configuration in full detail is shown in Appendix B, as well as the accompanying repository [8].

## 5.5 Further experiments

### 5.5.1 Training Format Comparison

The following experiments use the reduced hyperparameter configuration to compare the two proposed training formats introduced in Section 3.2.2. In particular, the experiments evaluate whether training the model to generate Jena Syntax Expressions or SPARQL queries leads to differences in retrieval performance and runtime. Table 18 summarizes the results. For brevity, only Assignment F1 is reported. The complete evaluation metrics are available in the accompanying software repository. The *Exec %* column denotes the share of scored items for which a non-empty query result was obtained. This metric provides an additional indication of the difference between successful query execution and obtaining an answer that matches the gold answer. The Llama-2 retraining experiments are included as reference values for comparison with the preceding experiments. The corresponding Qwen2.5 experiments use the reduced retrieval configuration described above, while keeping the remaining experimental conditions equivalent between the Jena and SPARQL variants. Note that the *Exec %* value of the Llama-2-13B CWQ SPARQL experiment includes stale items that were still processed as resolved. This experiment was produced with an earlier version of the retrieval script that did not skip stale items. Consequently, its reported execution rate is not directly comparable to the other experiments and should be considered an overestimate.

| Run | Exec % | F1 | time | Note |
|---|---|---|---|---|
| Llama-2-13b_CWQ-sparql | 94.0% | 76.5 | 18.28 s | SPARQL retrain |
| CWQ sparql | 85.3% | 74.4 | 5.55 s | Reduced config |
| CWQ jena | 84.7% | 74.0 | 8.75 s | Reduced config |
| Llama-2-13b_CWQ-sparql Gold | 96.3% | 83.8 | 4.33 s | SPARQL retrain |
| CWQ sparql Gold | 92.4% | 82.2 | 2.31 s | Reduced config |
| CWQ jena Gold | 91.9% | 81.8 | 2.58 s | Reduced config |
| Llama-2-7b_WebQSP-sparql | 97.9% | 76.3 | 15.56 s | SPARQL retrain |
| WebQSP sparql | 97.5% | 76.1 | 1.76 s | Reduced config |
| WebQSP jena | 95.8% | 74.5 | 2.70 s | Reduced config |
| Llama-2-7b_WebQSP-sparql Gold | 99.1% | 81.0 | 5.95 s | SPARQL retrain |
| WebQSP sparql Gold | 97.8% | 80.3 | 1.02 s | Reduced config |
| WebQSP jena Gold | 96.9% | 78.9 | 1.70 s | Reduced config |

**Table 18:** Comparison of training format performance impact under otherwise equal conditions. Retrain experiments are shown for reference values.

The table shows that the experiments using the SPARQL training target consistently achieve higher F1 scores than the corresponding experiments using Jena Syntax Expressions. The difference is relatively small on CWQ, where the Jena variant scores 0.4 percentage points lower than the SPARQL variant. On WebQSP, the difference is larger, with the Jena variant scoring 1.6 percentage points lower than the SPARQL variant. Additionally, the SPARQL experiments have a lower average runtime per item across both benchmarks. Compared to the earlier retraining experiments, the overall metrics are comparable, although the Qwen2.5-7B SPARQL variants achieve slightly lower F1 scores. On WebQSP, the difference is only 0.2 percentage points, while on CWQ the difference is 2.1 percentage points. These retraining experiments serve only as reference values, however, since both the hyperparameter configuration and the base model differ. This distinction is particularly relevant for CWQ, where the retraining experiment uses Llama-2-13B, whereas the reduced-configuration experiments use Qwen2.5-7B. The reduced hyperparameter configuration also substantially decreases the average runtime per item compared with the retraining experiments. The SPARQL variant is approximately 3.3 times faster on CWQ and 8.8 times faster on WebQSP.

### 5.5.2 Experiments on Wikidata

In this section, we evaluate the performance of the ChatKBQA methodology on QALD-7 [22], QALD-10 [23], WikiWebQuestions (WWQ) [30] and LC-QuAD 2.0 [9]. The split sizes for each of these benchmarks is specified in Appendix C. For the LC-QuAD 2.0 experiment, the reported Exec % excludes resolved stale items. The other three experiments were conducted using an older version of the retrieval algorithm that did not exclude such items.

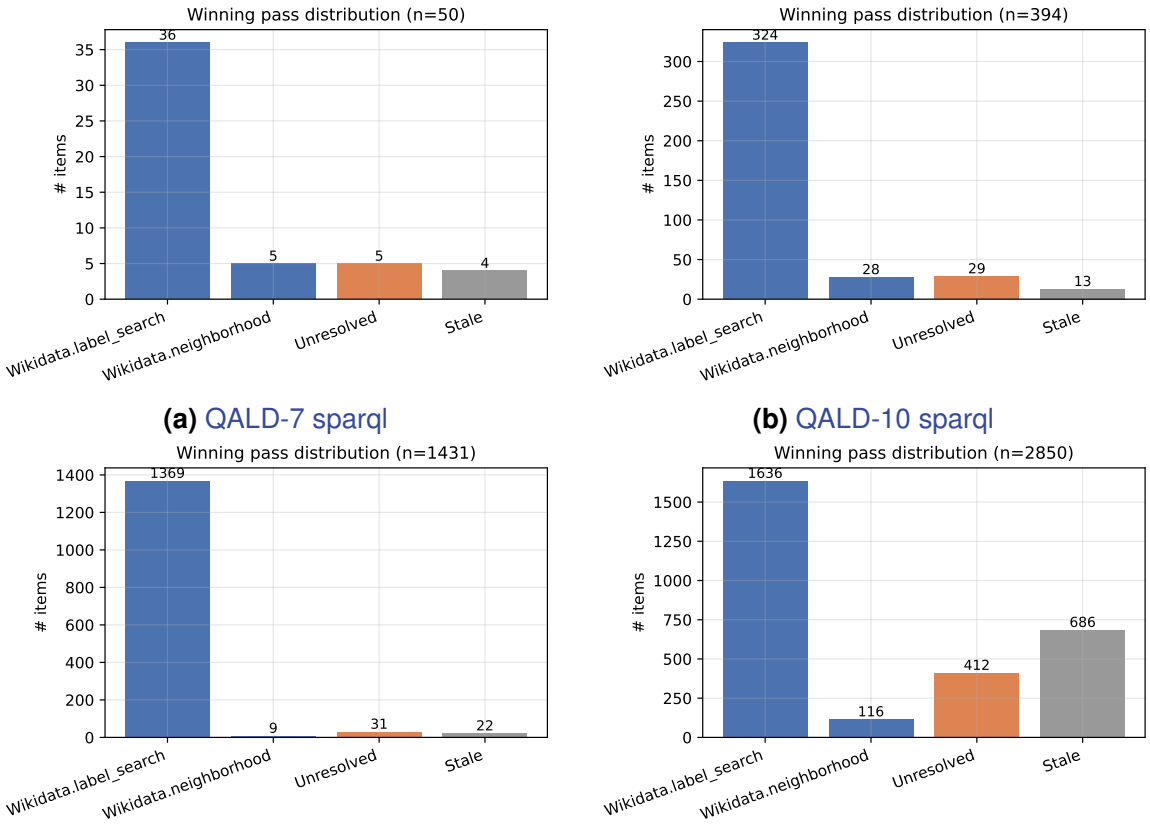
| Run | Exec % | EM | F1 | Hit@1 | time |
|---|---|---|---|---|---|
| QALD-7 sparql | 90.0% | 30.4 | 35.0 | 45.6 | 47.31 s |
| QALD-10 sparql | 91.9% | 33.9 | 34.9 | 36.0 | 46.94 s |
| WWQ sparql | 97.7% | 66.5 | 72.9 | 79.6 | 5.28 s |
| LC-QuAD 2.0 sparql | 81.0% | 59.7 | 62.0 | 63.7 | 30.46 s |

**Table 19:** Performance of ChatKBQA extension on various Wikidata benchmarks.

The results show substantial differences in performance across the evaluated benchmarks. WikiWebQuestions achieves the highest F1 and Hit@1 scores, with 72.9 and 79.6 percent, respectively. In contrast, QALD-7, QALD-10 and LC-QuAD 2.0 achieve F1 scores of 35.0, 34.9 and 62.0 percent. The average runtime also varies considerably between the datasets, ranging from 5.28 seconds per item on WikiWebQuestions to 47.31 seconds on QALD-7.

### 5.5.3 Runtime and Performance of Wikidata experiments

Figure 12 shows which predicate linking pass resolved how many items for the set of experiments specified in Table19.

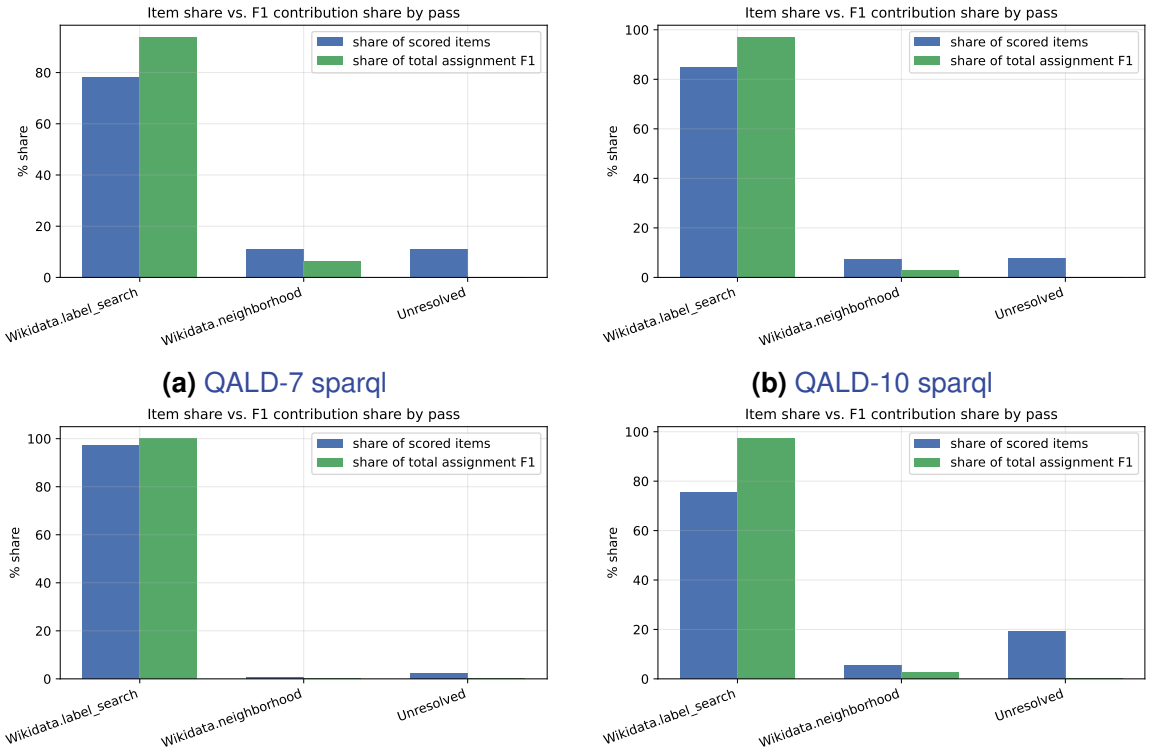



**(a)** QALD-7 sparql **(b)** QALD-10 sparql

**(c)** WWQ sparql **(d)** LC-QuAD 2.0 sparql

**Figure 12:** Distribution of items resolved by the different retrieval passes on Wikidata experiments.

The vast majority of items are either resolved by the Label Search Linker, unresolved or stale. Only a small fraction of items are resolved by the Neighborhood Linker. To get a better perspective on the actual contribution of the neighborhood linker, Figure 14 shows the share of items a specific predicate linking pass resolved, next to the share of the total Assignment F1 score it contributed.




**(a)** QALD-7 sparql **(b)** QALD-10 sparql

**(c)** WWQ sparql **(d)** LC-QuAD 2.0 sparql

**Figure 14:** Share of items resolved and share of total Assignment F1 contribution per predicate linking pass on Wikidata experiments.

Much like the Freebase experiments, the Label Search Linker resolves a large share of items. The graphs show that, among all experiments, the Neighborhood Linker resolves the most items on QALD-7, then QALD-10, then LC-QuAD 2.0, with almost no items resolved on WikiWebQuestions. The contribution to total Assignment F1 score follows a similar pattern.

### 5.5.4 Performance comparison to GRISP

In this section, we compare the metrics produced by this extension to the metrics provided by GRISP [27]. A comparison with GRISP is especially interesting, because it follows the same generate-then-retrieve approach, while also employing beam search during its inference phase. Furthermore, the training hyperparameters and base model used by GRISP are comparable to the setup we use in our reduced configuration (see Appendix B). The comparison results are summarized in Table 20.

| System | CWQ | WebQSP | QALD-7 | QALD-10 | WWQ |
|---|---|---|---|---|---|
| GRISP | **77.0** | 73.3 | **46.7** | **44.5** | **78.8** |
| ChatKBQA | 74.4 | **76.3** | 35.0 | 34.9 | 72.9 |

**Table 20:** Assignment F1 comparison between GRISP using Qwen2.5-7B and the ChatKBQA extension.

There is a clear difference in the metrics between the two methods. While GRISP records an Assignment F1 score 2.6 percentage points higher than ChatKBQA on CWQ, ChatKBQA's WebQSP Assignment F1 score exceeds GRISP's by 3.0 percentage points. However, on Wikidata benchmarks GRISP achieves higher Assignment F1 scores consistently. On QALD-7, the difference amounts to 11.7 percentage points. For QALD-10, GRISP scores 9.6 percentage points more than ChatKBQA. On WWQ, GRISP again achieves an Assignment F1 score 5.9 percentage points higher.

## 5.6 Error Analysis

In order to better understand the weak points of ChatKBQA, we analyze the performance of certain ChatKBQA components on WebQSP. For this, we run three experiments. The first experiment uses regular prediction beam generation and regular entity and predicate retrieval. It serves as the baseline and represents the performance of the standard ChatKBQA method. For the second experiment, we use regular prediction beam generation together with a set of Gold linkers instead of the regular linkers. These gold linkers only use the entities and predicates used in the gold SPARQL of any given dataset item and insert them into the prediction beam. This is an experiment with regular beam generation but perfect entity and predicate linking in the context of each item's gold SPARQL. Lastly, we run an experiment in which the regular prediction beam generation is replaced by a single prediction beam. This prediction beam is exactly the gold SPARQL with its entity and predicate identifiers substituted for their corresponding labels. This prediction beam is then processed using regular entity and predicate retrieval, to isolate the impact of beam generation on the final results. Table 21 shows this setup schematically.

| Experiment | Generation | Entity Linking | Predicate Linking |
|---|---|---|---|
| Baseline | Regular beam | Regular | Regular |
| Gold Linking | Regular beam | Gold | Gold |
| Gold Generation | Gold SPARQL | Regular | Regular |

**Table 21:** Experimental setup used for the WebQSP error analysis.

Table 22 shows the results of the error analysis. In the table, baseline refers to WebQSP sparql.

| Experiment | Assignment F1 | △ F1 |
|---|---|---|
| Baseline | 76.1 | – |
| Gold Linking | 74.3 | -1.8 |
| Gold Generation | 95.1 | +19 |

**Table 22:** Results of the WebQSP error analysis. △ F1 denotes the difference in Assignment F1 compared to the baseline.

The results show a substantial difference between the regular and Gold Generation experiments. Replacing regular beam generation with the gold SPARQL query increases Assignment F1 from 76.1% to 95.1%, corresponding to an improvement of 19.0 percentage points. In contrast, replacing regular entity and predicate linking with Gold linkers does not improve the result in this experimental setup. The Gold Linking experiment achieves an Assignment F1 of 74.3%, which is 1.8 percentage points below the baseline.

# 6 Discussion

The experiments presented in this work provide several insights into both the effectiveness and limitations of the ChatKBQA method. In particular, the results show that the original approach can be reproduced with comparable performance, that the retrieval configuration has a substantial influence on runtime, and that the generation and retrieval components contribute differently to the final performance. The experiments on Wikidata further demonstrate that the approach can be transferred to a different knowledge graph, although the performance depends strongly on the dataset and the available retrieval methods.

## 6.1 Reproduction of ChatKBQA

The reproduction experiments show that the extended implementation is able to reproduce the performance of the original ChatKBQA implementation within a relatively small margin. The largest difference between the original and reproduced results is 1.5 percentage points on CWQ and 2.4 percentage points on WebQSP. This performance loss on WebQSP specifically can also not solely be attributed to changes in the entity and predicate retrieval implementation, as the evaluation procedure differs from the original. These differences indicate that the implementation and evaluation pipeline used in this work behave sufficiently similarly to the original implementation to serve as a basis for the other experiments.

The retraining experiments further show that the general approach remains effective when using the extended training pipeline. The retrained models achieve results close to those reported for the original models, although the exact results differ between datasets and model configurations. In particular, the results cannot be interpreted as a direct comparison, since the original uses Logical Forms, while the retraining experiments use SPARQL as the training format. Instead, they establish the full extended pipeline implementation as a suitable basis for further experiments.

## 6.2 Retrieval Search Space

The hyperparameter experiments reveal a substantial amount of overprovisioning in the original retrieval configuration. The winning candidate distributions show that most successful entity and predicate candidates occur at indices that are significantly below the original configuration's maximum value. This is particularly apparent for the Neighborhood predicate linker, whose configuration permits up to 300 candidates for CWQ and 4000 candidates for WebQSP. This large search space has a substantial impact on runtime. On WebQSP, for example, the resolved Neighborhood queries

require substantially more time on average than the Simple queries. Furthermore, a small number of unresolved items dominates the total runtime because the system continues to process a large number of candidate combinations.

The sensitivity experiments indicate that the retrieval parameters can be reduced considerably without causing much loss in Assignment F1. Based on these observations, the reduced configuration decreases the retrieval search space greatly, while only sacrificing at most 2.1 percentage points in the case of CWQ. Furthermore, this Assignment F1 loss could partially be attributed to the base model change from Llama-2-13B to Qwen2.5-7B. It should however be noted that this configuration was selected heuristically rather than through formal hyperparameter optimization. Consequently, it should be regarded as a practical reduction of the search space rather than as an optimal configuration.

## 6.3 SPARQL and Jena Representations

The controlled comparison between the SPARQL and Jena training targets shows only relatively small differences in Assignment F1. The SPARQL representation performs slightly better than the Jena representation on all four evaluated configurations, with differences between 0.4 and 1.6 percentage points. At the same time, the SPARQL representation results in lower retrieval runtime. Additionally, there is no clear performance loss when using SPARQL as the training format as opposed to the original ChatKBQA's Logical Forms. As shown in Table 12, the SPARQL-based retrain experiments either match or surpass the original Logical Form predictions in all but one experiment, where the Logical Forms score higher. All in all, this result suggests that a special training format, like the original ChatKBQA's Logical Forms, does not contribute much to the overall performance, considering the effort needed to define their syntax. A special training format, unless it has a complete SPARQL parser, also hinders compatibility with other datasets or knowledge bases, because it inherently simplifies SPARQL queries and cannot reasonably express all possible SPARQL constructs, unless the custom syntax explicitly defines a way to represent them. At that point, the project moves away from a simplified SPARQL format and becomes a user-friendly version of SPARQL itself.

## 6.4 Transfer to Wikidata

The experiments on Wikidata demonstrate that the general approach can be transferred beyond Freebase. The system achieves high execution rates on all four evaluated datasets, indicating that a large proportion of the generated queries can be executed successfully against the Wikidata knowledge graph. However, the Assignment F1 scores vary considerably between datasets. In particular, ChatKBQA achieves substantially higher Assignment F1 on WikiWebQuestions than on the two QALD datasets, while

LC-QuAD 2.0 lies between them. This difference suggests that the difficulty of the retrieval and generation task depends strongly on the dataset.

ChatKBQA uses executability as a proxy for answer correctness. It can be reasoned that this proxy is considerably more accurate on Freebase due to the comparative sparseness of the knowledge graph. Wikidata is a substantially denser knowledge graph, which reduces the statistical chance of some arbitrary query targeting a group of entities producing the one correct query. Each Wikidata experiment's executability rate is considerably higher than the Assignment F1 score achieved. This is especially pronounced for QALD-7 and QALD-10, where executability is more than 50% higher than the resulting metrics. This indicates that the retrieval algorithm finds an executable query for almost any item in the dataset, while many of them are incorrect. This is also influenced by the general complexity of the gold SPARQL queries in both benchmarks, but Wikidata's density could also contribute to this. The same pattern can be seen for WikiWebQuestions. While executability rates on Freebase often fall into the 80-95% range, WikiWebQuestions has an executability rate of 97.7%, while the achieved Assignment F1 score is lower than what Freebase experiments with a similar executability rate achieve.

## 6.5 Comparison with GRISP

The comparison with GRISP shows that the results differ depending on the benchmark. On CWQ, GRISP reports a higher Assignment F1, whereas the ChatKBQA extension reports a higher value on WebQSP. On the Wikidata benchmarks, GRISP reports considerably higher Assignment F1 scores on all three datasets included in the comparison. These differences demonstrate that transferring the ChatKBQA approach to Wikidata does not by itself reproduce the performance of other generate-then-retrieve approaches. At the same time, the comparison does not identify a single cause for the observed differences. GRISP and the implementation presented in this work differ in several aspects of their retrieval and inference procedures, and the experiments conducted here were not designed as a controlled component-by-component comparison with GRISP.

## 6.6 Generation and Retrieval

The error analysis provides a further indication of where the limitations of the approach arise. Replacing the regular entity and predicate linkers with Gold linkers does not improve the final result in the evaluated setup. Instead, the produced metrics are actually worse. This indicates that restricting the retrieval candidates to entities and predicates appearing in the gold SPARQL does not necessarily improve the result and may remove alternative candidates that are useful for resolving the generated beams. In

contrast, replacing the regular prediction beam with the gold SPARQL query increases Assignment F1 from 76.1% to 95.1%. This large difference suggests that errors in the generated logical forms remain an important limitation of the overall pipeline. Within this experimental setup, improving the prediction beams appears to have a substantially larger effect on Assignment F1 than further refinement of the retrieval components.

## 6.7 Limitations and Future Work

The experiments also reveal several limitations of the current implementation. One important limitation concerns the runtime of the retrieval process. As shown in the runtime analysis, a small number of unresolved items can dominate the total runtime because the system continues to evaluate a large number of candidate combinations. An early stopping mechanism could therefore substantially reduce the runtime of the method. For example, prediction beams that repeatedly produce execution errors could be discarded after a certain number of unsuccessful attempts. However, defining a reliable criterion for identifying unpromising beams remains an open problem, as a beam that initially produces execution errors may still contain useful candidates in later retrieval steps. A second limitation is that the reduced retrieval configuration was selected heuristically based on the observed candidate distributions and sensitivity analysis. It was not obtained through a systematic hyperparameter optimization. The configuration therefore represents a practical reduction of the original search space rather than an optimal parameter setting. Future work could investigate automated methods for selecting retrieval parameters based on the characteristics of individual datasets or prediction beams. The Wikidata experiments also leave room for further improvements to the retrieval component. The entity and predicate linkers used in this work were designed to be similar in concept to their Freebase counterparts and do not necessarily provide optimal coverage of the Wikidata schema. Additional retrieval strategies could therefore be investigated, particularly methods that exploit the structural characteristics of Wikidata more directly. Finally, the error analysis suggests that improving query generation is an important direction for future work. The large performance increase obtained when using the gold SPARQL query indicates that a considerable portion of the remaining errors originates before the retrieval stage. Future work could therefore investigate methods for improving the quality and diversity of the generated prediction beams.

The infrastructure developed in this work also provides a basis for transferring the approach to additional knowledge bases. Since the knowledge-base interaction and retrieval components are separated from the main pipeline, support for another knowledge base can be added by implementing suitable entity and predicate linkers, as well as an appropriate Knowledge Base module. Future work could therefore investigate

the transfer of the extended ChatKBQA approach to other knowledge bases beyond Freebase and Wikidata.

# 7 Conclusion

In this work, we expand on the existing ChatKBQA project. We transform it into a modular system of entity and predicate linkers to define specific retrieval behaviors. The addition of Knowledge Base modules removes significant implementation barriers when adding support for a different knowledge base. We conducted a series of experiments and established comparability between the original implementation and this extended implementation. Furthermore, we reduce the hyperparameter configuration proposed by ChatKBQA. The new configuration produces comparable results while recording a significant decrease in runtime. Using it, we establish baseline values on common Wikidata benchmarks.

Several insights are provided by this work. The concrete choice of training format has a relatively small impact in the context of the experiments conducted. In particular, directly generating SPARQL performs comparably to the Jena Syntax Expressions while providing a more direct connection between the generated output and the final query executed against the knowledge base.

The transfer to Wikidata demonstrates that the general ChatKBQA approach works beyond Freebase. At the same time, the results show that successful query execution does not necessarily imply correct answers, particularly on the Wikidata benchmarks. This highlights the importance of adapting the retrieval process to the characteristics of the target knowledge base. The error analysis further indicates that query generation remains an important limitation of the overall approach, as providing the gold query leads to a substantially higher performance than regular generation.

Overall, this work provides a modular extension of ChatKBQA, demonstrates its applicability to Wikidata, and establishes initial baseline results for its application to several Wikidata benchmarks. The resulting infrastructure also provides a basis for future extensions to other knowledge bases and for further improvements to both query generation and retrieval.

# Acknowledgements

First, I would like to thank my supervisor, Sebastian Walter, for his support, thoughtful assistance, and answers to my questions throughout this work. I would also like to thank Prof. Hannah Bast for agreeing to examine my thesis.

I would also like to thank all the people who created and maintain the software that this work depends on. In particular, I would like to thank the ChatKBQA team for making their work publicly available.

Further thanks go to the Corporate Design team at the Albert-Ludwigs-Universität Freiburg and Leander Marius Bürkin for providing this LaTeX template.

# List of Figures


1 Simple representation of a single RDF triple. . . . . 3
2 A possible RDF knowledge graph. Square vertices indicate literals, while oval vertices indicate IRIs. . . . . 3
3 Overview of the ChatKBQA architecture [15] . . . . . 8
5 Distribution of items resolved by the different retrieval passes on retrain experiments. . . . . 39
7 Exact Match, Assignment F1 and Hit@1 per predicate linker pass on retrain experiments. . . . . 40
9 Share of resolved items and total Assignment F1 score contribution per predicate linker pass on retrain experiments. . . . . 41
10 Hyperparameter sensitivity per predicate linker pass for retrain experiments. 43
12 Distribution of items resolved by the different retrieval passes on Wikidata experiments. . . . . 48
14 Share of items resolved and share of total Assignment F1 contribution per predicate linking pass on Wikidata experiments. . . . . 49
16 Distribution of winning index for the Simple Linker. The winning index is expressed as a combination of winning beam, entity permutation and predicate permutation index. . . . . XII
18 Distribution of winning index for the Neighborhood Linker. The winning index is expressed as a combination of winning beam, entity permutation and predicate permutation index. . . . . XIII

# List of Tables


1 Result of SPARQL query defined in Code 1 if it were executed against the RDF graph shown in Figure 2. . . . 4
2 Syntax of the Logical Form representation used by the original ChatKBQA approach. . . . 10
3 Parameters of the extended entity and predicate retrieval procedure. . . . 19
4 Inputs provided to entity and predicate linkers. . . . 23
5 Outputs returned by entity and predicate linkers. . . . 24
6 Parameters of the neighborhood predicate linker. . . . 26
7 Parameters of the WBSearchEntities linker. . . . 28
8 Knowledge-base-specific functionality provided by the KB module. . . . 29
9 Variables used in the complexity analysis. . . . 31
10 Computational resources allocated to the retrieval experiments. . . . 33
11 Comparison of the reproduced ChatKBQA results with the reported results. 35
12 Comparison of adapted prediction results with the retrain results. . . . 36
13 Hyperparameter setup used by the retraining experiments, taken from original ChatKBQA retrieval code [15]. . . . 37
14 Distribution of items resolved and average retrieval runtime by predicate linking pass. . . . 40
15 Summary statistics of the winning candidate indices for each retrain experiment and predicate linker pass. EP denotes the entity permutation index, PP the predicate permutation index, and Beam the beam rank. . 42
16 Reduced retrieval hyperparameter configuration. . . . 44
17 Entity and predicate linker configurations used for the different knowledge bases and gold settings. . . . 44
18 Comparison of training format performance impact under otherwise equal conditions. Retrain experiments are shown for reference values. . . . 46
19 Performance of ChatKBQA extension on various Wikidata benchmarks. 47
20 Assignment F1 comparison between GRISP using Qwen2.5-7B and the ChatKBQA extension. . . . 50
21 Experimental setup used for the WebQSP error analysis. . . . 50
22 Results of the WebQSP error analysis. $\Delta$ F1 denotes the difference in Assignment F1 compared to the baseline. . . . 51
24 Full comparison table between using the label fallback and not using the label fallback. . . . XI
25 Train and test set sizes of the benchmarks used in this work. . . . XV

# List of Algorithms


1 Abstract SPARQL to retrieve all entities of type city. . . . . 4
2 Example SPARQL query taken from WebQSP [31] . . . . . 9
3 Example ChatKBQA Logical Form . . . . . 9
4 Example Jena Syntax Expression . . . . . 11
5 Example of a normalized SPARQL query . . . . . 11
6 Type map entry for the Country entity . . . . . 12
7 Sample group beam search inference output (5 candidate SPARQL queries) 13
8 Example SPARQL query without label fallback . . . . . 16
9 Example SPARQL query with label fallback . . . . . 16
10 Example of adapting an original ChatKBQA prediction to the extended Freebase representation. . . . . 34
11 Training configuration used for further experiments. . . . . XIV

# Glossary

| Notation | Description | Page List |
|---|---|---|
| Candidate | A knowledge base entity or predicate retrieved as a possible substitute for a specific mention. | 15, V |
| Candidate Set | The set of all Candidates for a single mention. | 15, V |
| Entity Mention | A placeholder in a Prediction Beam that is intended to be linked to an actual knowledge base entity during retrieval. | 15, V |
| Entity Permutation | A variant of a Prediction Beam in which each Entity Mention has been substituted by exactly one Candidate from its corresponding Candidate Set. | 15, V |
| Predicate Mention | A placeholder in a Prediction Beam that is intended to be linked to an actual knowledge base predicate during retrieval. | 17, V |
| Predicate Permutation | A variant of an Entity Permutation in which each Predicate Mention has been substituted by exactly one Candidate from its corresponding Candidate Set. | 15 |
| Prediction Beam | A single raw model prediction containing Entity Mentions and Predicate Mentions in the model's training format. | 15, V |

# References


[1] Apache Software Foundation. Apache Jena, 2026. Accessed September 19, 2026.

[2] Apache Software Foundation. Apache Jena: SPARQL S-Expressions, 2026. Accessed September 19, 2026.

[3] Tom Ayoola, Shubhi Tyagi, Joseph Fisher, Christos Christodoulopoulos, and Andrea Pierleoni. ReFinED: An efficient zero-shot-capable approach to end-to-end entity linking. In Anastassia Loukina, Rashmi Gangadharaiah, and Bonan Min, editors, *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Track*, pages 209–220, Hybrid: Seattle, Washington + Online, July 2022. Association for Computational Linguistics.

[4] Hannah Bast and Björn Buchhold. Qlever: A query engine for efficient sparql+ text search. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 647–656, 2017.

[5] Hannah Bast, Johannes Kalmbach, Robin Textor-Falconi, and Christoph Ullinger. Sparqloscope: A generic benchmark for the comprehensive and concise performance evaluation of sparql engines. In Daniel Garijo, Sabrina Kirrane, Angelo Salatino, Cogan Shimizu, Maribel Acosta, Andrea Giovanni Nuzzolese, Sebastián Ferrada, Thibaut Soulard, Kouji Kozaki, Hideaki Takeda, and Anna Lisa Gentile, editors, *The Semantic Web – ISWC 2025*, pages 22–40, Cham, 2026. Springer Nature Switzerland.

[6] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1247–1250, New York, NY, USA, 2008. Association for Computing Machinery.

[7] The UniProt Consortium. Uniprot: the universal protein knowledgebase in 2025. *Nucleic Acids Research*, 53(D1):D609–D617, 01 2025.

[8] Luis Drayer. Bachelor's thesis: Porting chatkbqa to wikidata, 2026. https://github.com/Ludraaa/Bachelorarbeit.

[9] Mohnish Dubey, Debayan Banerjee, Abdelrahman Abdelkawi, and Jens Lehmann. Lc-quad 2.0: A large dataset for complex question answering over wikidata and dbpedia. In *International semantic web conference*, pages 69–78. Springer, 2019.

[10] Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). 06 2013.

[11] Tianyu Gao, Xingcheng Yao, and Danqi Chen. SimCSE: Simple contrastive learning of sentence embeddings. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 6894–6910, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics.

[12] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3c recommendation, W3C, March 2013.

[13] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[14] Hugging Face. transformers-community/group-beam-search, 2026. Hugging Face model repository, accessed September 19, 2026.

[15] Haoran Luo, Haihong E, Zichen Tang, Shiyao Peng, Yikai Guo, Wentai Zhang, Chenghao Ma, Guanting Dong, Meina Song, Wei Lin, Yifan Zhu, and Anh Tuan Luu. ChatKBQA: A generate-then-retrieve framework for knowledge base question answering with fine-tuned large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics ACL 2024*, pages 2039–2056, Bangkok, Thailand and virtual meeting, August 2024. Association for Computational Linguistics.

[16] Haklay M. and Weber P. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, 10 2008.

[17] OpenLink Software. Virtuoso universal server documentation, 2026. Accessed September 19, 2026.

[18] Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2025.

[19] Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *The Twelfth International Conference on Learning Representations*, 2024.

[20] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. In Marilyn Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 641–651, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[21] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[22] Ricardo Usbeck, Axel-Cyrille Ngonga Ngomo, Bastian Haarmann, Anastasia Krithara, Michael Röder, and Giulio Napolitano. 7th open challenge on ques-

tion answering over linked data (qald-7). In Mauro Dragoni, Monika Solanki, and Eva Blomqvist, editors, *Semantic Web Challenges*, pages 59–69, Cham, 2017. Springer International Publishing.

[23] Ricardo Usbeck, Xi Yan, Aleksandr Perevalov, Longquan Jiang, Julius Schulz, Angelie Kraft, Cedric Möller, Junbo Huang, Jan Reineke, Axel-Cyrille Ngonga Ngomo, Muhammad Saleem, and Andreas Both. Qald-10 – the 10th challenge on question answering over linked data: Shifting from dbpedia to wikidata as a kg for kgqa. *Semantic Web*, 15(6):2193–2207, 2024.

[24] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[25] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledge base. *Communications of the ACM*, 57:78–85, 2014.

[26] Sebastian Walter and Hannah Bast. Grasp: Generic reasoning and sparql generation across knowledge graphs. In Daniel Garijo, Sabrina Kirrane, Angelo Salatino, Cogan Shimizu, Maribel Acosta, Andrea Giovanni Nuzzolese, Sebastián Ferrada, Thibaut Soulard, Kouji Kozaki, Hideaki Takeda, and Anna Lisa Gentile, editors, *The Semantic Web – ISWC 2025*, pages 271–289, Cham, 2026. Springer Nature Switzerland.

[27] Sebastian Walter and Hannah Bast. Grisp: Guided recurrent iri selection over sparql skeletons. *arXiv preprint arXiv:2604.21133*, 2026.

[28] Wikimedia Foundation. Wikidata: Statistics, 2026. Accessed September 19, 2026.

[29] World Wide Web Consortium. RDF 1.1 Concepts and Abstract Syntax. W3c recommendation, W3C, February 2014.

[30] Silei Xu, Shicheng Liu, Theo Culhane, Elizaveta Pertseva, Meng-Hsi Wu, Sina Semnani, and Monica Lam. Fine-tuned LLMs know more, hallucinate less with few-shot sequence-to-sequence semantic parsing over Wikidata. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on*

*Empirical Methods in Natural Language Processing*, pages 5778–5791, Singapore, December 2023. Association for Computational Linguistics.

[31] Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. The value of semantic parse labeling for knowledge base question answering. In Katrin Erk and Noah A. Smith, editors, *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206, Berlin, Germany, August 2016. Association for Computational Linguistics.

[32] Jing Zhang, Xiaokang Zhang, Jifan Yu, Jian Tang, Jie Tang, Cuiping Li, and Hong Chen. Subgraph retrieval enhanced model for multi-hop knowledge base question answering. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 5773–5784, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[33] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pages 400–410, 2024.

# Appendix

# A Additional Results

## A.1 Impact of label fallback on metrics

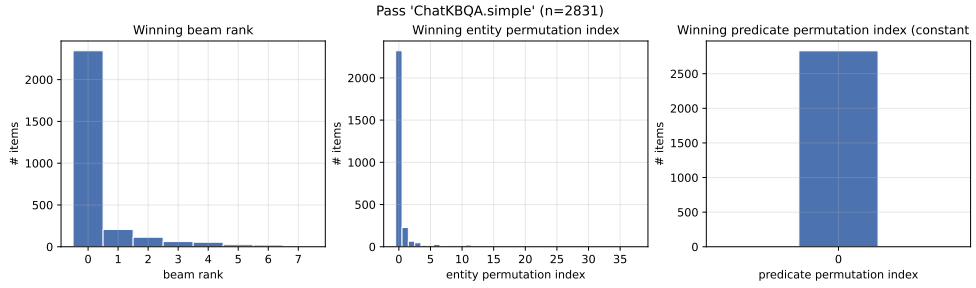
Table 24 compares the evaluation metrics with and without label fallback for the experiments where both configurations are available. The original ChatKBQA results are included for reference.

| Run | EM | F1 | Hit@1 | Label Fallback |
|---|---|---|---|---|
| Original ChatKBQA CWQ | 73.3 | 77.8 | 82.7 | Yes |
| Original ChatKBQA CWQ Gold | 76.8 | 81.3 | 86.0 | Yes |
| Reproduction CWQ | 71.7 | 75.8 | 80.0 | No |
| Reproduction CWQ | 74.3 | 78.3 | 82.7 | Yes |
| Reproduction CWQ Gold | 78.3 | 82.2 | 86.3 | No |
| Reproduction CWQ Gold | 78.3 | 82.2 | 86.3 | Yes |
| Llama-2-13b CWQ-sparql | 72.8 | 76.5 | 80.5 | No |
| Llama-2-13b CWQ-sparql | 75.3 | 79.0 | 83.0 | Yes |
| Llama-2-13b CWQ-sparql Gold | 80.5 | 83.8 | 87.5 | No |
| Llama-2-13b CWQ-sparql Gold | 80.5 | 83.9 | 87.6 | Yes |
| Original ChatKBQA WebQSP | 73.8 | 79.8 | 83.2 | Yes |
| Original ChatKBQA WebQSP Gold | 77.8 | 83.5 | 86.4 | Yes |
| Reproduction WebQSP | 73.0 | 77.5 | 82.0 | No |
| Reproduction WebQSP | 73.3 | 77.9 | 82.4 | Yes |
| Reproduction WebQSP Gold | 76.5 | 81.0 | 85.3 | No |
| Reproduction WebQSP Gold | 76.7 | 81.1 | 85.3 | Yes |
| Llama-2-7b WebQSP-sparql | 72.0 | 76.3 | 80.5 | No |
| Llama-2-7b WebQSP-sparql | 72.2 | 76.6 | 80.9 | Yes |
| Llama-2-7b WebQSP-sparql Gold | 77.0 | 81.0 | 85.3 | No |
| Llama-2-7b WebQSP-sparql Gold | 76.5 | 80.5 | 84.8 | Yes |

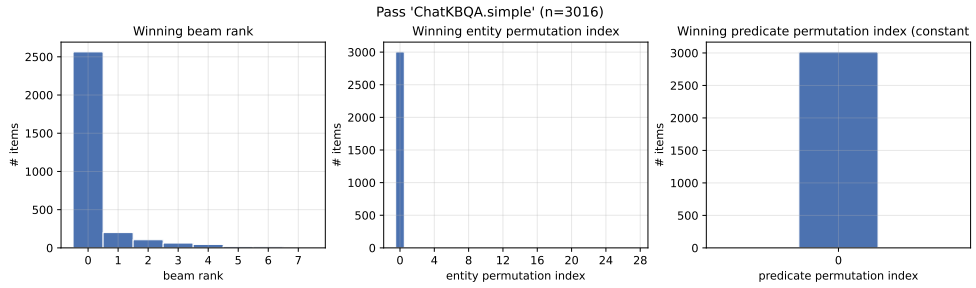
**Table 24:** Full comparison table between using the label fallback and not using the label fallback.

## A.2 Full winning index distribution in retrain experiments

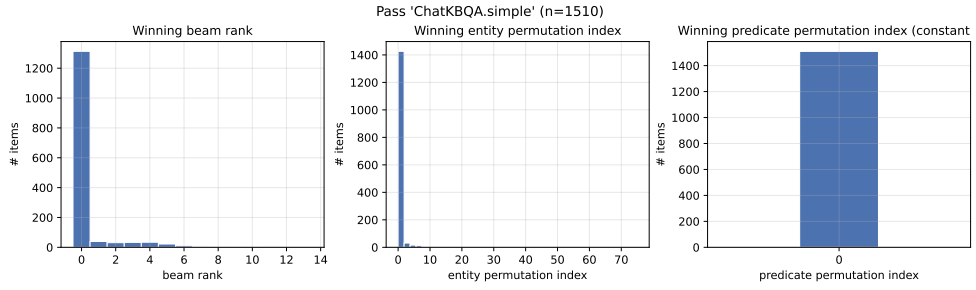
### A.2.1 Simple



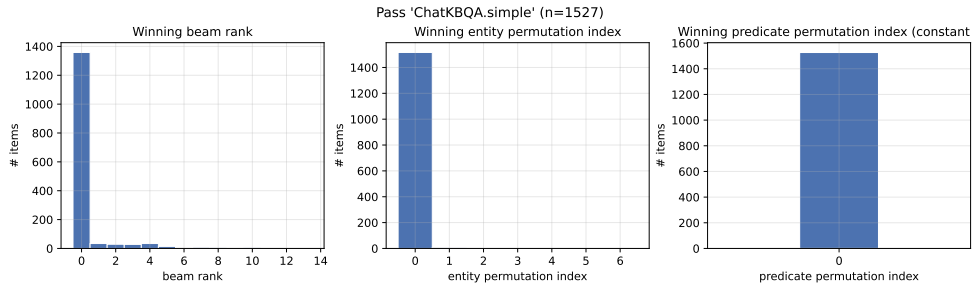

**(a)** CWQ




**(b)** CWQ Gold



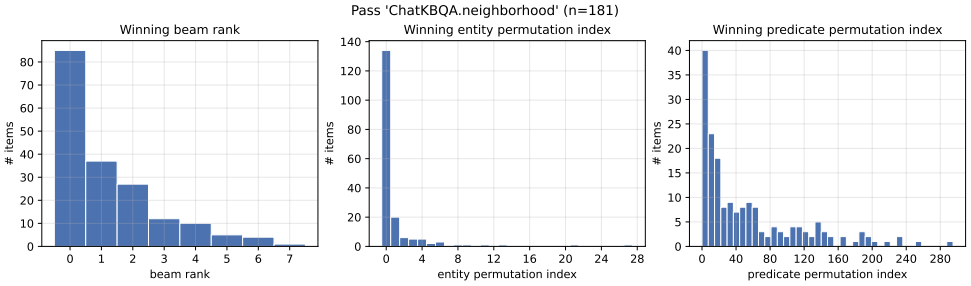

**(c)** WebQSP



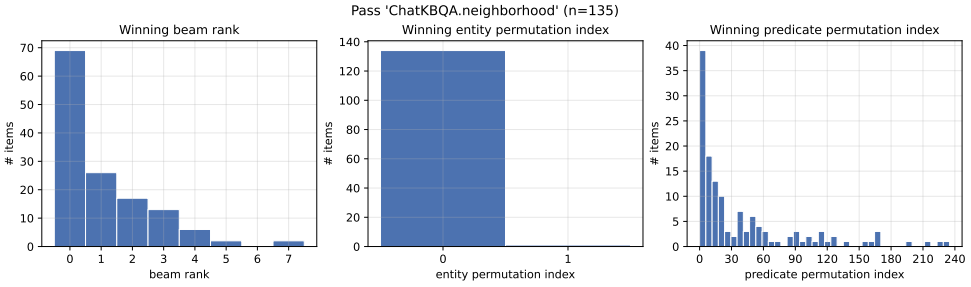

**(d)** WebQSP Gold

**Figure 16:** Distribution of winning index for the Simple Linker. The winning index is expressed as a combination of winning beam, entity permutation and predicate permutation index.

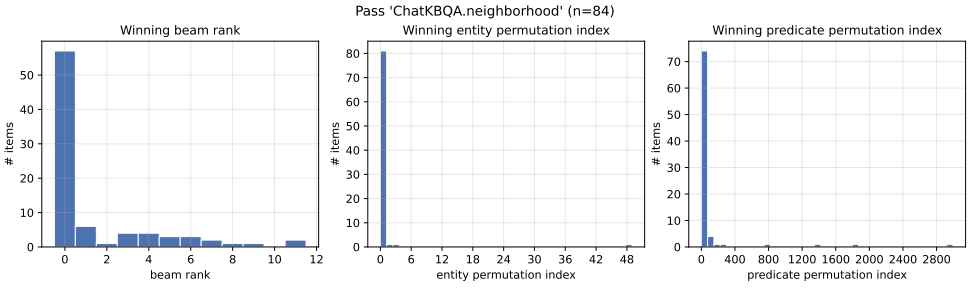
### A.2.2 Neighborhood



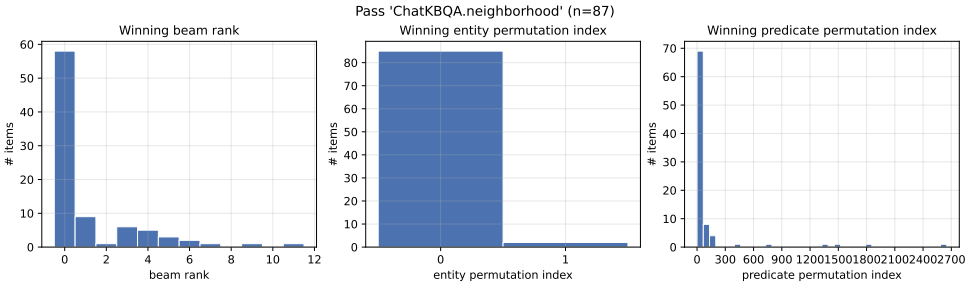

**(a)** CWQ




**(b)** CWQ Gold




**(c)** WebQSP




**(d)** WebQSP Gold

**Figure 18:** Distribution of winning index for the Neighborhood Linker. The winning index is expressed as a combination of winning beam, entity permutation and predicate permutation index.

# B Training Configuration

All experiments, except for ones reproducing ChatKBQA, use the training configuration specified below for fine-tuning. Notably, this uses Qwen-2.5-7B as the base model, as opposed to the Llama 2 family. Note *val_size*, *eval_steps* and *save_steps* were sometimes modified to accommodate either very small or very large training datasets.

The complete training config for every experiment is available in the accompanying repository [8].

```
### Model
model_name_or_path: LLMs/Models/Qwen25-7b/

### Method
stage: sft
finetuning_type: lora

### LoRA config
lora_target: all
lora_rank: 32
lora_alpha: 32
lora_dropout: 0.05

### Dataset
template: qwen
cutoff_len: 2048

### Eval
val_size: 0.05
per_device_eval_batch_size: 2
eval_strategy: steps
eval_steps: 1000

### Output
save_steps: 1000
load_best_model_at_end: true
metric_for_best_model: eval_loss
greater_is_better: false

per_device_train_batch_size: 1
gradient_accumulation_steps: 8
learning_rate: 1e-4
num_train_epochs: 4.0
lr_scheduler_type: cosine
warmup_ratio: 0.1
fp16: true
weight_decay: 0.01
max_grad_norm: 1.0
```

**Code 11:** Training configuration used for further experiments.

# C Benchmark Split Size

| Benchmark | Train | Test |
|---|---|---|
| CWQ | 27,639 | 3,531 |
| WebQSP | 3,453 | 1,639 |
| QALD-7 | 100 | 50 |
| WWQ | 2,431 | 1,431 |
| QALD-10 | 412 | 394 |
| LC-QuAD 2.0 | 24,124 | 6,028 |

**Table 25:** Train and test set sizes of the benchmarks used in this work.

*Note:* Due to time constraints, evaluation on LC-QuAD 2.0 was restricted to the first 2,850 test samples.