

**Hochschule Darmstadt**

– Fachbereich Mathematik–

**Generation of Meaningful SQL-Query  
Exercises Using Large Language Models and  
Knowledge Graphs**

Abschlussarbeit zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

**Paul Christ**

Matrikelnummer: 771982

Referent : Prof. Dr. Markus Döhring

Korreferent : Prof. Dr. Torsten Munkelt



## DECLARATION

---

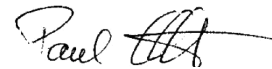
Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

*Rüsselsheim, 20. Januar 2023*



---

Paul Christ

## ABSTRACT

---

This thesis presents a system for generating meaningful SQL-query-exercises for educational purposes. The system utilizes a combination of relational database concepts, knowledge graphs and natural language generation to generate exercises that are both syntactically correct and semantically meaningful. The system allows for parametrization of the SQL-query-exercise generation algorithm, to provide the user with fine granular control over the included SQL-query concepts. The exercise task consists of the formulation of a SQL-query, based on the information presented in the database schema and the natural language description of the generated SQL-query. The resulting exercises are designed to teach students the skills necessary to effectively retrieve data from a database using SQL, while reducing manual effort in creating SQL-query-exercises by hand. The results are evaluated by a crowd-working framework and show a promising foundation in the generation of meaningful SQL-query-exercises.

## ZUSAMMENFASSUNG

---

In dieser Arbeit wird ein System zur Generierung von semantisch plausiblen SQL-Query-Übungen für Ausbildungszwecke vorgestellt. Das System nutzt eine Kombination aus relationalen Datenbankkonzepten, Wissensgraphen und natürlicher Sprachgenerierung, um Übungen zu generieren, die sowohl syntaktisch korrekt als auch semantisch sinnvoll sind. Das System erlaubt die Parametrisierung des Algorithmus zur Generierung von SQL-Abfragen, um dem Benutzer eine fein granulare Kontrolle über die enthaltenen SQL-Abfragekonzepte zu ermöglichen. Die Übungsaufgabe besteht aus der Formulierung einer SQL-Abfrage, basierend auf den Informationen des Datenbankschemas und der natürlichsprachlichen Beschreibung der generierten SQL-Abfrage. Die daraus resultierenden Übungen sollen den Studierenden die notwendigen Fähigkeiten vermitteln, um Daten aus einer Datenbank mit Hilfe von SQL effektiv abzurufen und gleichzeitig den manuellen Aufwand für die Erstellung von SQL-Abfrageübungen zu reduzieren. Die Ergebnisse werden mit Hilfe eines Crowdfunding-Frameworks evaluiert und zeigen eine vielversprechende Grundlage bei der Generierung sinnvoller SQL-Abfrageübungen.

# CONTENTS

---

## I THESIS

1	INTRODUCTION	2
1.1	Motivation . . . . .	2
1.2	Aim and scope . . . . .	2
1.3	Methodology . . . . .	3
2	ASPECTS OF RELATIONAL DATABASES, NATURAL LANGUAGE PROCESSING AND KNOWLEDGE GRAPHS	4
2.1	Aspects of Relational Databases Regarding the Generation of SQL-Queries . . . . .	4
2.1.1	Related Work in Generating SQL-Query Exercises . . . . .	4
2.1.2	Assessing the Complexity of SQL-Queries . . . . .	5
2.2	Natural Language Processing for Transforming SQL to Natural Language . . . . .	6
2.2.1	Related Work in Generating Text from SQL-Queries . . . . .	6
2.2.2	Natural Language Generation Pipeline . . . . .	7
2.3	Knowledge Graphs for Semantically Enriching Relational Databases	8
2.3.1	Current Landscape of Knowledge Graphs . . . . .	8
2.3.2	Entities of the Knowledge Graph as Tables . . . . .	9
3	ANALYSIS OF THE REQUIREMENTS FOR A SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES	11
3.1	Features of the Exercise Generation Algorithm . . . . .	11
3.1.1	Parametrization of the Exercise Generation System . . . . .	11
3.1.2	Generation of an Exercise Formulation in Natural Language . . . . .	11
3.1.3	Automatic Assessment of the Solution Attempts . . . . .	12
3.2	Consistency and Quality of the Generated Exercises . . . . .	12
3.2.1	Syntactic Soundness of the Generated Exercise . . . . .	12
3.2.2	Semantic Plausibility of the Generated Exercise . . . . .	12
3.2.3	Unambiguity of the Exercise Formulation Description in Natural Language . . . . .	13
3.3	Properties of the Exercise Generation System . . . . .	13
3.3.1	Constant Access and Readiness and Platform Independent Usability of the Exercise Generation System of the Exercise Generation System . . . . .	13
3.3.2	Performance of the Exercise Generation System . . . . .	13
4	DESIGN AND IMPLEMENTATION OF A SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES	14
4.1	Overview of a System for the Generation of Meaningful SQL-Query Exercises . . . . .	14
4.2	Deriving a Relational Database from a Knowledge Graph . . . . .	15

4.2.1	Subsetting a Knowledge Graph Into Topically Coherent Domains . . . . .	15
4.2.2	Deriving a Relational Schema from Entities of the Knowledge Graph . . . . .	16
4.2.3	Limitations of Deriving Relational Databases From Knowledge Graphs . . . . .	17
4.3	Semantically Enriching a Relational Database with a Knowledge Graph . . . . .	18
4.3.1	Normalizing the Relational Database to Satisfy Domain Constraints and Cardinality Restrictions . . . . .	18
4.3.2	Knowledge Graph Entities for Semantic Labeling of Tables . . . . .	19
4.4	Generation of Meaningful SQL-Query Exercises . . . . .	22
4.4.1	Parameter Space of the Generation Algorithm . . . . .	22
4.4.2	Traversing the Relational Schema as a Graph of Tables and Foreign Key Constraints . . . . .	22
4.4.3	Generation of Random Parameter-Compliant SQL-Queries . . . . .	24
4.5	Generation of an SQL-Query Exercise Formulation in Natural Language . . . . .	25
4.5.1	Macroplanning the SQL-Query Exercise Formulation . . . . .	25
4.5.2	Microplanning the SQL-Query Exercise Formulation . . . . .	26
4.5.3	Surface Realization of the Structured Exercise Formulation . . . . .	26
5	EVALUATION OF THE SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES . . . . .	29
5.1	Evaluation study design . . . . .	29
5.1.1	Crowdsourcing for Evaluating the Generated Exercise Component . . . . .	29
5.2	Evaluation of the Exercise Generation Algorithm Features . . . . .	31
5.2.1	Evaluation of the Generation of an Exercise Formulation in Natural Language . . . . .	31
5.2.2	Evaluation of the Automatic Assessment of the Solution Attempts . . . . .	32
5.3	Evaluation of the Consistency and Quality of the Generated SQL-Query Exercises . . . . .	32
5.3.1	Evaluation of the Syntactic Soundness of the Generated Exercise . . . . .	32
5.3.2	Evaluation of the Semantic Plausibility of the Generated Exercise . . . . .	32
5.4	Evaluation of the SQL-Query Exercise Generation System Properties . . . . .	33
5.4.1	Evaluation of the Performance of the Exercise Generation System . . . . .	33
6	CONCLUSION AND FUTURE WORK . . . . .	36
6.1	Conclusion . . . . .	36
6.2	Future Work . . . . .	36

**II APPENDIX**

<b>A APPENDIX</b>	<b>39</b>
A.1 DQL Keyword Subset . . . . .	39
A.2 PostgreSQL Reflection Queries . . . . .	41
A.2.1 PostgreSQL Table Reflection Query . . . . .	41
A.2.2 PostgreSQL Foreign Key Reflection Query . . . . .	41
A.3 NLG Template Repositories . . . . .	42
A.3.1 Baseline NLG Templates . . . . .	42
A.3.2 Hybrid NLG templates . . . . .	43
<b>BIBLIOGRAPHY</b>	<b>46</b>



## LIST OF FIGURES

---

Figure 2.1	Near impossible query translation case as presented in [61] . . . . .	7
Figure 2.2	Natural Language Generation (NLG)-Pipeline according to [58] . . . . .	7
Figure 4.1	Unified Modeling Language (UML) component diagram of the SQL-query exercise generation system . . .	14
Figure 4.2	Wikidata hyperonymy graph of the entities 'voluntary association' and 'sports club', that are both types of 'organization' . . . . .	17
Figure 4.3	Normalised relational model of the IMDB database . . .	18
Figure 4.4	Handcrafted IMDB knowledge graph . . . . .	20
Figure 4.5	Subfloat - Figure . . . . .	23
Figure 4.6	SQL-query Abstract Syntax Tree (AST) example . . . . .	25
Figure 4.7	Subfloat - Figure . . . . .	27
Figure 4.8	Complex SQL-query AST . . . . .	27
Figure 4.9	Subfloat - Figure . . . . .	28
Figure 5.1	Assessment opening screen . . . . .	29
Figure 5.2	General questions about the SQL-query . . . . .	30
Figure 5.3	Error categories per NLG-approach . . . . .	30
Figure 5.4	Human-likeness of the SQL-query descriptions . . . . .	31
Figure 5.5	Subfloat - Figure . . . . .	31
Figure 5.6	Subfloat - Figure . . . . .	33
Figure 5.7	Subfloat - Figure . . . . .	34
Figure 5.8	Runtime of generating a SQL-query exercise without outliers . . . . .	35
Figure 5.9	Runtime of generating a SQL-query exercise with outliers . . . . .	35

## LIST OF TABLES

---

Table 2.1	Table structure modelled by cardinalities . . . . .	9
Table 4.1	Query generation algorithm parameters . . . . .	21

## LIST OF ALGORITHMS

---

Figure 1	Map Wikipedia categories to Wikidata entities . . . . .	16
Figure 2	Sort knowledge graph entities into semantically similar tables . . . . .	16
Figure 3	Table selection by generating a subgraph of the database schema . . . . .	22

## LISTINGS

---

Listing 4.1	SQL-query generation order . . . . .	24
Listing 4.2	Numeric constraint operators . . . . .	24
Listing 4.3	String constraint operators . . . . .	24
Listing 4.4	SQL-query string example . . . . .	25
Listing 4.5	SQL-query description structuring . . . . .	25
Listing 4.6	Complex SQL-query string example . . . . .	27
Listing A.1	Data Query Language (DQL) keyword subset . . . . .	39
Listing A.2	Table reflection query . . . . .	41
Listing A.3	Foreign Key reflection query . . . . .	41
Listing A.4	Baseline SQL-constituent templates . . . . .	42
Listing A.5	Hybrid SQL-constituent templates . . . . .	43

## ACRONYMS

---

UML	Unified Modeling Language
DQL	Data Query Language
DDL	Data Definition Language
DML	Data Manipulation Language
CFG	Context Free Grammar
GUI	Graphical User Interface
DAG	Directed Acyclic Graph
LSTM	Long Short Term Memory
NLG	Natural Language Generation
SPARQL	SPARQL Protocol and RDF Query Language
LCA	Lowest Common Ancestor
AST	Abstract Syntax Tree
LLM	Large Language Model

Part I  
THESIS

## INTRODUCTION

---

### 1.1 MOTIVATION

Students and professors are often confronted with a limited supply of available exercises, as creating new exercises is a time-consuming process. Some of these exercises are commonly used as teaching examples or solved together in practical courses, leaving students with little to no exercises for solidifying the underlying concepts in self study. This limited pool of exercises also concerns the construction of exams, often resulting in not publishing sample exams, to not disclose potential exam exercises.

Furthermore, existing exercises may be too difficult or too easy, or address only certain aspects of the task and thus fail to meet students at their current knowledge level [72]. In addition to the common lack, or insufficient detail of sample solutions and the limited availability of teaching staff to provide feedback regarding the subject, this may attribute to underperforming and less motivated students [20].

An increasing number of studies has shown that the aforementioned issues also apply to SQL-query exercises, such as a lack of practice opportunities [49], resignation in the case of unavailable feedback [50] and varying knowledge levels [56, 60].

In order to more effectively teach query formulation, educators should emphasize natural language patterns, query planning, and increasingly ambiguous exercises [67].

### 1.2 AIM AND SCOPE

This thesis aims to provide a system, that is capable of generating SQL-query exercises, consisting of a SQL-query and a natural language SQL-query description. The exercise goal is to produce a SQL-query that yields the same query-result as the originally generated SQL-query. The only input for fulfilling the exercise goal, is the database schema and the natural language description of the generated SQL-query. It is assumed, that a system for generating SQL-query exercises enables time and location independent self-study and alleviate the time expenditure on creating and curating SQL-query exercises. That said, the goal of this thesis is explicitly not to measure the educational effectiveness of the generated SQL-query exercises. Due to the expressiveness of SQL-queries only a subset of the DQL part of SQL is utilized.

### 1.3 METHODOLOGY

The goal of this work is to develop a system for generating meaningful SQL-query exercises, to reduce the expenditure of time on creating and curating said exercises manually and to enable students to self-study independent of time and location.

Due to the broad assortment of relevant disciplines involved in creating a system for generating SQL-query exercises, such as relational databases, natural language processing and graph theory, proven results of prior works will be integrated if applicable.

The evaluation of the proposed system for generating SQL-query exercises was performed by consulting human feedback. The evaluators were curated to be somewhat familiar with SQL-query formulation.

The data collection was performed at one single point in time, due to money and time constraints and no immediate reflux of information into the used methods was performed.



## ASPECTS OF RELATIONAL DATABASES, NATURAL LANGUAGE PROCESSING AND KNOWLEDGE GRAPHS

---

### 2.1 ASPECTS OF RELATIONAL DATABASES REGARDING THE GENERATION OF SQL-QUERIES

#### 2.1.1 *Related Work in Generating SQL-Query Exercises*

Of the entire field of automatic question generation, only the subfield of generating SQL-query-exercises is considered. Two major approaches apply to the automatic generation of SQL-queries. The first approach generates SQL-queries from natural language. With the recent success of large neural language models, a surge of papers utilizing this approach is currently recorded [11, 39, 57]. The second approach generates SQL-queries in a more constraint-oriented and structured rule-based approach. In the following only related works following the second approach are presented, as this thesis utilizes the second approach as well. Also general SQL-query teaching tools and frameworks, which do not generate the exercises, are omitted, as the focus of this work is in the generation of SQL-query-exercises instead of considering SQL-specific teaching techniques.

Gudivada et al. [21] provide a theoretical outline of adapting a Context Free Grammar (CFG) for arithmetic expressions to SQL queries. The approach is limited to a small subset of SQL keywords, predicates and operators. The generated queries are abstract in the sense that they are not tied to an existing database and that there is no natural language description provided. Due to this approach, the generated queries lack in plausibility, semantic depth and the ability to specify the query constituents.

Do et al. [12] propose a generic approach to the generation of SQL-query-exercises, which is based on database metadata of any database schema. The approach allows for a selection of an extensive list of SQL-constituents that are to be used in the SQL-query and allows for JOIN-statements to cross multiple tables. The user is then prompted to explain the generated SQL-query according to predefined questions, but not to recreate the query bases on alternative output, like the SQL-query result or a natural language description of the query. This approach generates SQL-queries that lack in plausibility and semantic depth.

Atcharyachanvanich et al. [4] propose the so called *RSQLG*-algorithm. The algorithm generates a SQL-query first and then a natural language description based on the generated SQL-query. The algorithms approach to generating SQL-queries is similar to [12], in that it generates a SQL-query based on the metadata of any database schema. Compared to [12], the algorithm operates on a smaller set of SQL-constituents and supports the inclu-

sion of only one table at a time. The generated natural language description is a direct mapping of The approach allows for specifying which constituents are used in the SQL-query generation, but the generated SQL-queries still lack in plausibility.

The approach presented in [4] has been extended by Dwivedi et al. [13] to support Data Definition Language (DDL) commands and by Rakesh et al. [10] to support DDL and Data Manipulation Language (DML) commands. Basse et al. [6] extend the proposed approach in [4] by allowing predefined JOIN-statements for up to three tables. This extension requires adding manual metadata in an ontology, which makes the generation algorithm no longer generic.

### 2.1.2 Assessing the Complexity of SQL-Queries

*Complexity* is an ambiguous concept whose concrete definition is dependent on the context domain. Complexity science can be seen as the *study of the phenomena which emerge from a collection of interacting objects* [35]. Multiple emergent phenomena can be observed, when applying this definition to SQL-queries. Examples of emergent phenomena are computation time, memory requirement or intricacy of control flow. This thesis focuses on the emergent phenomenon of perceived difficulty of writing a SQL-query from the perspective of somebody learning SQL.

Thus, SQL-query complexity is defined as follows: *The complexity that results in the cognitive effort required to formulate or comprehend the SQL-query.*

A number of complexity measures to approximate the cognitive effort required to formulate or comprehend a piece of software in general have been introduced over the years [24, 46, 65, 66, 73, 81]. These metrics are mostly tied to the occurrences of certain concepts or keywords contained in a piece of code, but usually do not consider an individual weighing of the concepts or keywords. As multiple studies have shown, especially novices experience different levels of difficulty depending on the concepts that are used [2, 3, 48–50, 56, 60, 63, 67, 69].

Furthermore, different error types have been examined, such as syntax errors, semantic errors and logic errors. [56, 67, 69] report that syntax errors are observed less than semantic and logic errors. [47, 50, 60, 69] on the other hand report the opposite, but all report that students are mostly able to fix syntax errors by themselves, whereas semantic and logic errors often lead to abandoning the query when no hints are given. In the case of semantic and logic errors, students are observed to transform their initial query into unnecessarily complex constructs by incremental changes with low word-based edit distances [50]. Students appear to struggle with queries including *JOINS*, *GROUP BY* statements and *subqueries* the most [2, 3, 48, 63] Thus, complexity measures to approximate the cognitive effort required to formulate SQL-queries ideally should include a way of associating custom weights to specific concepts or SQL-constituents.

In upcoming mentions of the complexity of SQL-queries, the above definition is referred to, if not specifically mentioned otherwise.

The effect of database schema complexity on the generation of SQL-queries [68] is explicitly not considered.

## 2.2 NATURAL LANGUAGE PROCESSING FOR TRANSFORMING SQL TO NATURAL LANGUAGE

### 2.2.1 *Related Work in Generating Text from SQL-Queries*

The approaches for SQL-to-text can be categorized into two different classes: 1.) template- and rule-based and 2.) neural machine translation. Rule-based approaches explicitly employs the information represented by the query, along with predefined text-templates, to create a corresponding natural language description. Neural machine translation is usually used for translating a natural language into another natural language, but can also be utilized to transform a structured language into natural language or vice versa. This typically involves a large amount of training data, but doesn't require the manual construction of transformation rules.

Koutrika et al, [38] propose a rule-based approach, by transforming the SQL-query into a directed graph and formulating generic text-templates for edges and manually created text-templates for paths along the graph. Additional meta-information about the database schema has to be provided by a human modeler. In order to provide concise natural languages descriptions of the SQL-queries, a lot of effort is required for every database schema. The approach is extensible for the entire SQL syntax and always provides correct descriptions in terms of SQL-query-content.

Kokkalis et al. [37] extend the approach of [38] by adding support for the generation of multilingual SQL-query-descriptions, as well as providing a Graphical User Interface (GUI) for browsing and annotating database schemes.

Eleftherakis et al. [15] expanded upon the system introduced in [37] by extending the translation capability to support more SQL-query-constituents and the fluency of the generated description by modifying the template mechanism.

Iyer et al. [33] introduce a neural sequence-to-sequence model for translating SQL-queries to a natural language description. The SQL-queries are relatively simple and only feature one table at a time and simple constraint-clauses. The proposed model utilizes the Long Short Term Memory (LSTM) architecture for encoding the SQL-query and a feed forward neural network for decoding the natural language description. It also is able to generate natural language descriptions of C# code. While the model generates textual descriptions in an end-to-end fashion, it only produces correct descriptions ~ 63% of the time.

Xu et al. [80] introduce a neural graph-to-sequence model for translating a Directed Acyclic Graph (DAG) representation of simple SQL-queries to a nat-

ural language description. The proposed model consists of a graph encoder and an attention based sequence decoder. By encoding the graph structure and thus the neighbours of each SQL-constituent node, it produces correct descriptions  $\sim 75\%$  of the time.

Ma et al. [43] expand on the approach presented in [80] by utilizing the transformer architecture [74]. The model was trained and tested on more complex SQL-queries, that feature subqueries and multiple JOINS. The adaptability to more complex SQL-queries is achieved by encoding the SQL-query structure with custom attention strategies. While the model produces correct descriptions only  $\sim 66\%$  of the time, it does so with considerably more complex queries than the approaches of [33, 80].

**Query 9.** In query *Q9*, the semantics is unclear as well, but in a rather different way. This time, syntactically, one does see the **all** connector as the main challenge, but the expression ‘= all’ will have to be interpreted as ‘earliest’ in this case, which is very difficult to obtain.

```

select a.name
from MOVIES m, CAST c, ACTOR a
where m.id = c.mid and c.aid = a.id
and year <= all (
  select m1.year
  from MOVIES m1, MOVIES m2
  where m1.title = m.title and m2.title = m.title
  and m1.id != m2.id
)
    
```

Consequently, it is very difficult to produce the following text:

*“Find the actors who have played in the earliest versions of movies that have been repeated”*

Figure 2.1: Near impossible query translation case as presented in [61]

As seen, both rule-based and neural machine translation approaches still possess major limitations, due to the complexity of the SQL-to-text task. Simitsis et. al [61] present a set of cases that illustrate the complexity of the SQL-to-text task even further. Figure 2.1 shows one extreme such case.

### 2.2.2 Natural Language Generation Pipeline

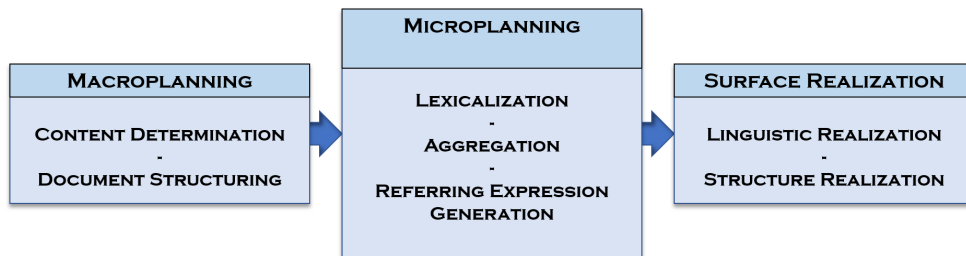


Figure 2.2: NLG-Pipeline according to [58]

The standard **NLG** pipeline is a widely accepted architecture, originally proposed by Reiter et al. [58]. As depicted in figure 2.2 it is composed of three major components, that are traversed from left to right and top to bottom: 1.) Macroplanning, 2.) Microplanning and 3.) Surface Realization.

Macroplanning consists of two subtasks: a) **content determination** determines the text content and b) **document structuring** organises the high level structuring of, e.g. paragraphs or sections.

Microplanning consists of three subtasks: a) **lexicalization** selects appropriate expressions and words to use, b) **aggregation** organises the low level structuring of, e.g. sentences and c) **referring expression generation** identifies the entities and their references.

Surface Realization consists of two subtasks: a) **linguistic realization** applies grammar and syntax rules to the abstract text and b) **structure realization** converts the abstract representation into eventually required mark-up symbols for the presentation component (such as XML, for example).

Not every **NLG**-pipeline requires all steps. Which steps may be used, depends on the input data and target output or environment.

A recent development is also to break up end-to-end neural **NLG** models to substitute certain aspects of the pipeline depicted in figure 2.2 or use this pipeline as a preprocessing step for the end-to-end models [16]. This allows for more explainability and fine-grained control in the end-to-end models and less manual work in creating the components of the **NLG**-pipeline shown in figure 2.2 [78].

## 2.3 KNOWLEDGE GRAPHS FOR SEMANTICALLY ENRICHING RELATIONAL DATABASES

### 2.3.1 Current Landscape of Knowledge Graphs

There exists a large number of definitions for knowledge graphs that are not congruent. Early mentionings of knowledge graphs define it as "A mathematical structure with vertices as knowledge units connected by edges that represent the prerequisite relation." [44], "A particular kind of semantic network." [55] and "A graph that understands real-world entities and their relationships to one another: things, not strings." [62]. More recent definitions are: "A knowledge graph acquires and integrates information into an ontology and applies a reasoner to derive new knowledge" [14], "A data model used in the Semantic Web [...] based on three basic principles: 1. Encode knowledge using statements. 2. Express background knowledge in ontologies. 3. Reuse knowledge between datasets" [79] and "A knowledge graph is presented as the intersection of the formal models able to represent facts of various types and levels of abstraction using a graph-based formalism." [75].

It is therefore unclear what exactly classifies a knowledge graph. For the purpose of this work a knowledge graph must consist of:

1. entities that are classified by types,
2. directed binary semantic relationships between entities and

3. the specific relationships:

- a) *is-a*-relationships between types (hyponymy),
- b) *has-a*-relationships between entities (meronymy) and
- c) *instance-of*-relationships between entities and instances.

This definition excludes semantic networks such as WordNet [51], common sense knowledge graphs such as OpenCyc [42], TransOMCS, ConceptNet [64] and ATOMIC [29], as they do not contain *instance-of*-relationships.

Eligible knowledge graphs that are considered further are YAGO [70], DBpedia [41] and Wikidata [77]. Knowledge graphs that are subsets of any of the aforementioned knowledge graphs, such as ASER [83] or Freebase [71], are not considered individually.

While there have been efforts to comparatively analyse knowledge graphs on several metrics, no conclusive "*best of breed*" has been determined as of yet [1, 17, 32, 53, 54]. While YAGO, DBpedia and Wikidata are being generally similar regarding their content size, depth and width, they most notably differ in a) their datasource and b) their underlying ontology. The content in DBpedia is derived from Wikipedia articles and the underlying ontology is generated from a manually specified mapping, that maps the extracted content of wikipedia articles to their respective DBpedia types [41]. Wikidata instead derives its content from multiple sources, including Freebase, digital libraries and manual user input. The underlying ontology is community built in a bottom-up fashion [77]. YAGO imports its content directly from Wikidata, but uses the Schema.org [22] ontology instead [70].

As knowledge graphs grow in size, modifications to the underlying ontology are necessary to accommodate domain changes and their natural evolution over time. Currently only the ontology specified by Wikidata follows a transparent versioning scheme with fine-grained changes due to the bottom-up approach [23, 36].

### 2.3.2 Entities of the Knowledge Graph as Tables

In order to transform a knowledge graph into a relational database schema, the constituents need to be mapped accordingly.

Cardinality	Result
1-1	Table attribute
1-n	Lookup table
n-m	Junction table

Table 2.1: Table structure modelled by cardinalities

The entities of a knowledge graph can be translated to individual tables. By traversing the taxonomy, derived from the *is-a*-relationships, of an entity a common table name can be assigned. The meronymy, derived from the *has-a*-relationships, of an entity allows to assign properties and attributes to the

table. The cardinality of every has-a-relationship of an entity can be derived by iterating the instances of an entity and its respective has-a-relationships (how many Bs is A connected to) and the other way around (how many is As is B connected to). The same needs to be done for semantic relationships between entities to derive the cardinality. The cardinality can then be used to transfer table attributes into separate tables. As listed in table 2.1 the cardinalities require to model the following table structures: A 1-1 cardinality means that it is a proper table attribute. A 1-n cardinality means that a lookup-table is required. A n-m cardinality means that a junction-table is required.

## ANALYSIS OF THE REQUIREMENTS FOR A SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES

---

### 3.1 FEATURES OF THE EXERCISE GENERATION ALGORITHM

#### 3.1.1 *Parametrization of the Exercise Generation System*

The system shall allow users to parameterize the generation algorithm to regulate the complexity of the generated SQL-query and the selection of SQL-keywords in the generation process.

Applying the definition of complexity stated in chapter 2.1.2, the resulting complexity of a generated SQL-query is defined as being governed by the following criteria:

1. The number of distinct SQL-keywords that occur in the generated SQL-query.
2. The number of occurrences of a specific SQL-keyword in a generated SQL-query.
3. The number of distinct operands a specific SQL-keyword receives.
4. The total number of operands per SQL-keyword.

#### 3.1.2 *Generation of an Exercise Formulation in Natural Language*

For each generated SQL-query, the system shall formulate a description of the corresponding result set in *human-like* natural language.

The exercise description shall consist of a depiction of the underlying database schema and a description of the result set.

What makes machine-generated text *human-like* differs depending on the domain and task. The task-dependent evaluation of what makes a text *human-like* often lacks concrete specifications [40, 84]. Thus for the task of generating descriptions of SQL-queries, the following criteria are defined to determine the property of *human-like*:

1. Conciseness
  - a) Text length, for which shorter is preferred.
  - b) Information aggregation, for which less repetitions are preferred.
  - c) Technical details that are a result of database normalization, for which less are preferred.
2. Correctness



- a) Missing information, where less is preferred.
- b) Writing style, for which better is preferred.
- c) Comprehensibility, for which more is better.

The conciseness-criteria aim to reflect the general human trait to omit information that seems self-evident (tacit knowledge) [7], as well as the manner in which a set of data would be described in a practical setting, which is rather the question of *what* data is required instead of *how* the data should be extracted.

### 3.1.3 *Automatic Assessment of the Solution Attempts*

For every emitted solution attempt, the system shall be able to assess whether the solution is correct and in case of an error produce a score and a hint.

A solution attempt is defined as a syntactically correct SQL-query. In case of syntactical errors only the error message of the database system shall be propagated.

The correctness of a solution is to be determined by matching the result set instead of the queries, as the projection of SQL-queries to result sets is surjective but not injective.

In contrast, the score and hint generation is to be determined by measuring the distance between the solution attempt and the generated query, as almost identical queries can produce vastly different result sets.

## 3.2 CONSISTENCY AND QUALITY OF THE GENERATED EXERCISES

### 3.2.1 *Syntactic Soundness of the Generated Exercise*

The system shall generate SQL-queries that adhere to standard SQL syntax in general and specifically the DQL subset of SQL [30, 31].

Due to the vast space of possibilities the *SELECT*-command enables, the available keywords shall be further reduced to the subset defined in A.1.

To ensure syntactically correct SQL-queries, the keywords *SELECT* and *FROM* are required to occur exactly once, with at least one argument each. Furthermore all non-aggregate columns referenced in the select list are to be specified as grouping columns. The lower and upper bounds regarding the number of arguments for the remaining keywords must be determined dynamically depending on the database schema.

### 3.2.2 *Semantic Plausibility of the Generated Exercise*

The generated SQL-query shall be semantically plausible regarding *a)* the coherency of the combination of SQL-query-constituents and *b)* the actual meaning of the query objective.

The coherency of the generated SQL-query may be considered semantically plausible if:

1. the query doesn't contain redundant constituents,
2. the combination of constituents doesn't contradict the behaviour of one or more other constituents,
3. every constituent effects the result set.

The query objective may be considered semantically plausible if the query objective:

- is deemed logically feasible and
- adheres to selectional preference in the context of the database schema.

### 3.2.3 *Unambiguity of the Exercise Formulation Description in Natural Language*

The generated natural language SQL-query description shall be unambiguous.

This means there should be no information lost from the translation process of the generated SQL-query to the natural language description, when paired with the information of the database schema.

Given the database schema and the natural language description, then producing a SQL-query that generates the same output as the generated SQL-query should always be feasible.

## 3.3 PROPERTIES OF THE EXERCISE GENERATION SYSTEM

### 3.3.1 *Constant Access and Readiness and Platform Independent Usability of the Exercise Generation System of the Exercise Generation System*

The system shall be presented in such a way that allows for constant digital access and readiness and the ability to deploy the system on any platform.

### 3.3.2 *Performance of the Exercise Generation System*

The system shall perform the query-generation in a reasonable amount of time. A reasonable amount of time is declared as a run time that does not increase faster than a polynomial function of the input size.

## DESIGN AND IMPLEMENTATION OF A SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES

### 4.1 OVERVIEW OF A SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES

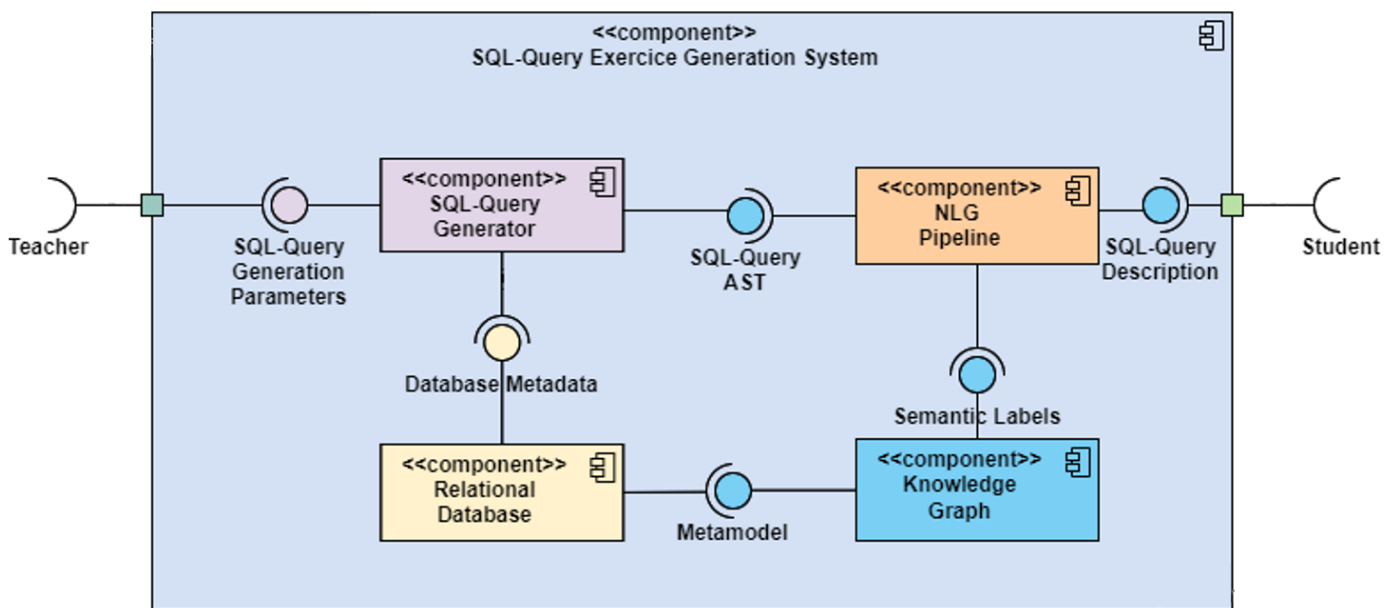


Figure 4.1: UML component diagram of the SQL-query exercise generation system

Figure 4.1 displays a highlevel overview over the SQL-query exercise generation system, in the scenario of a teacher generating an exercise for a student. The teacher gives a set of parameters as input to the SQL-query generator. The SQL-query generator utilizes database metadata of the relational database to generate the SQL-Query. The relational database in turn is enriched with additional semantic information via a metamodel provided by a knowledge graph. The NLG-pipeline handles the creation of a natural language description of the SQL-query, by using a SQL-query AST and the semantic labels provided by the knowledge graph. The resulting SQL-query description provided by the NLG-pipeline can then be consumed by the student.

## 4.2 DERIVING A RELATIONAL DATABASE FROM A KNOWLEDGE GRAPH

### 4.2.1 *Subsetting a Knowledge Graph Into Topically Coherent Domains*

The knowledge contained in Wikidata is considered to be the most comprehensive due to its multiple sources and the most conclusive due to the rapid and explicit changes in the ontology. Thus all following experiments concerning knowledge graphs are performed on Wikidata unless specified otherwise.

Due to the sheer size of the Wikidata knowledge graph, a single relational schema to contain all the information is not feasible. 1. schemas that are too large may hinder student learning [45, 82] 2. realistic db schemas may be large, but don't span across a large number of domains / are cohesive in their modelled domain

Generating subsets of Wikidata is a common use case, which commonly involve the use of the taxonomical structure of Wikidata entities, albeit in different ways. Different subsetting methods may be categorized into approaches based on a) query-languages [5, 9], b) patterns or entity-schemas [8, 18, 19] and c) graph-theoretic techniques [26, 76].

However, approaches of category a) and b) produce very narrow subgraphs that consist of a singular or small number of different entities and require extensive knowledge about the ontology of the general knowledge graph [34]. Approaches of category c) tend to produce subgraphs with more diverse entity compositions but less topical cohesion [28].

To avoid the shortcomings of the aforementioned approaches, a folksonomy, the Wikipedia Category Graph [25] is used to derive subgraphs that consist of multiple entities, while being topically cohesive. Wikimedia provides access to a mapping between Wikipedia pages, their categories and corresponding Wikidata entities <sup>1</sup>.

The Wikipedia category graph is processed and mapped to their Wikidata entities as portrayed in algorithm 1. The algorithm receives a list of the toplevel Wikipedia categories  $t$  and a flattened list of Wikipedia categories and their immediate subcategories or pages  $f$ . The toplevel categories are then iterated and used as their individual starting points for further nesting paths  $p$ . The categories below the toplevel are then added onto a stack  $s$ , that is iterated until empty and continuously filled with not yet visited, deeper nested categories. Whether or not a nested entity  $e$  is a category or a leaf node (a Wikipedia-page), is marked by a prefix. Regardless of whether it is a nested category,  $e$  is then added to a nested dynamic HashMap representation  $g$  of the current Wikipedia-category-subgraph at its corresponding location. If a Wikidata entity identifier can be associated with it, it is added as well. As the Wikipedia category graph may contain infinite loops, a list of already visited categories  $v$  in the current path has to be kept and checked against, to terminate the program.

---

<sup>1</sup> See pages database at: [https://www.mediawiki.org/w/index.php?title=Manual:Database\\_layout/diagram&action=render](https://www.mediawiki.org/w/index.php?title=Manual:Database_layout/diagram&action=render)

---

**Algorithm 1** Map Wikipedia categories to Wikidata entities

---

**Input:**  $t, f$ **Output:**  $g$ 

```

1:  $s \leftarrow \text{empty Stack}$ 
2:  $v \leftarrow \text{empty Set}$ 
3:  $g \leftarrow \text{empty HashMap}$ 
4: for  $c$  in  $t$  do
5:   push  $c$  on  $s$ 
6:    $p \leftarrow c$ 
7:   while  $s$  is not empty do
8:      $sc \leftarrow s.\text{pop}()$ 
9:     for  $e$  in  $f[sc]$  do
10:      if  $\text{ISCATEGORY}(e)$  then
11:        if  $e$  not in  $v$  then
12:           $s.\text{push}(e)$ 
13:        end if
14:      end if
15:       $g[sc] \leftarrow \text{WIKIDATAIDENTIFIER}(e)$ 
16:    end for
17:  end while
18: end for
    RETURN( $g$ )

```

---

## 4.2.2 Deriving a Relational Schema from Entities of the Knowledge Graph

---

**Algorithm 2** Sort knowledge graph entities into semantically similar tables

---

**Input:**  $E_s$ **Output:**  $T$ 

```

1:  $T \leftarrow \text{empty HashMap}$ 
2: for  $e \in E_s$  do
3:    $M_e \leftarrow \text{empty HashMap}$ 
4:   for  $\hat{e} \in E_s$  do
5:     if  $e \neq \hat{e}$  then
6:        $h \leftarrow \text{HYPERONYMYGRAPH}(e)$ 
7:        $\hat{h} \leftarrow \text{HYPERONYMYGRAPH}(\hat{e})$ 
8:        $H \leftarrow \text{UNION}(g, \hat{g})$ 
9:        $t \leftarrow \text{LOWESTCOMMONANCESTOR}(H)$ 
10:       $M_e[t].\text{add}(\hat{e})$ 
11:    end if
12:  end for
13:   $t \leftarrow \text{MAX}(M_e)$ 
14:   $T[t] \leftarrow M_e[t]$ 
15:   $E_s \leftarrow E_s \setminus M_e[t]$ 
16: end for
17: RETURN( $T$ )

```

---

In order to derive a relational schema from the entities of a category subgraph a pairwise Lowest Common Ancestor (LCA)-approach is utilized, as shown in algorithm 2. The algorithm gets the set of entities of the category subgraph  $E_s$  as input. Then the entities are shuffled, iterated pairwise and the LCA,  $t$  of their united hyperonymy-graph  $H$  is determined. The paired entity  $\hat{e}$  is then assigned to the set of entities with the paired LCA  $t$ . After all entities are paired, the  $t$  with the largest number of assigned entities is persisted as its own table. The entities assigned to  $t_{max}$  are then removed from  $E_s$ . After pairing all entities the algorithm returns the set of tables  $T$  with their associated entities.

### 4.2.3 Limitations of Deriving Relational Databases From Knowledge Graphs

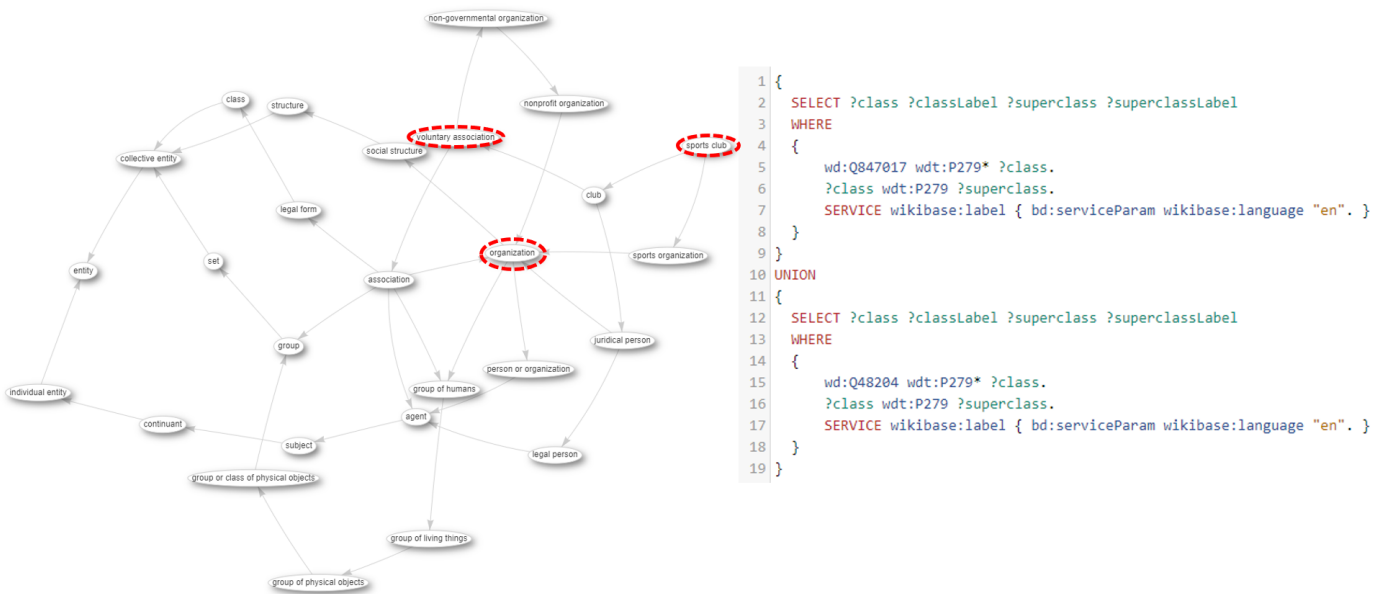


Figure 4.2: Wikidata hyperonymy graph of the entities ‘voluntary association’ and ‘sports club’, that are both types of ‘organization’

The produced relational databases are of low quality. This is mostly due to the fact, that the algorithm that infers a relational database schema fails to merge different entities along their hyperonymy-graph into sensible joint tables. Thus, either a very small number of very generic tables, consisting of very diverse instances, or a very large number of highly overlapping tables with only very few instances are created, depending on the cutoff that is set. In figure 4.2 one such case is highlighted. The SPARQL Protocol and RDF Query Language (SPARQL)-query on the right recursively gathers the hyperonymy-graph of the ‘voluntary association’- and ‘sports club’-entities. When their graphs are joined together, the lowest common node is ‘organisation’. While not being false, this is not the desired level of table granularity, especially considering, that the example was taken out of the mapped Wikidata entries to the subgraph of the Wikipedia basketball category.

Further it is unclear what attribute belongs to which class in the hyperonymy-system and how they are composed, as the properties are set on the instance-level instead of the class-level.

It is also ambiguous what entity is a class and what is an instance of a class, as this is largely dependent on the surrounding context, which is not encoded in Wikidata. This may result in entities being tables themselves, while also being a data row of another table.

The approach also fails at establishing clear cut offs to differentiate between additional entities and mere properties. Both of which may be connected to an entity that belongs to a domain subgraph, but cannot be classified as such.

### 4.3 SEMANTICALLY ENRICHING A RELATIONAL DATABASE WITH A KNOWLEDGE GRAPH

#### 4.3.1 Normalizing the Relational Database to Satisfy Domain Constraints and Cardinality Restrictions

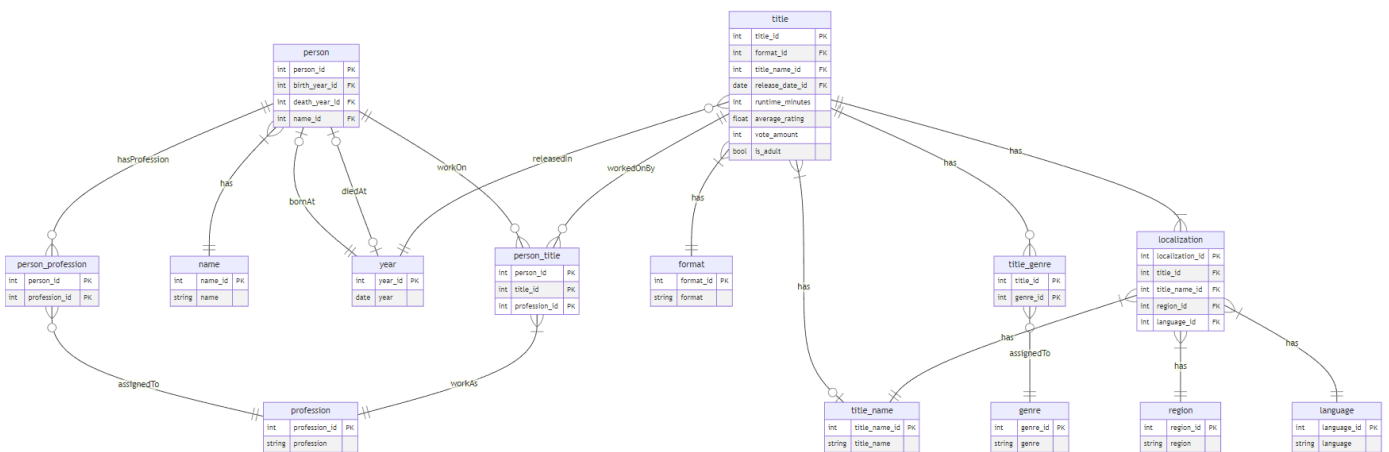


Figure 4.3: Normalised relational model of the IMDB database

Due to the issues in generating a relational database from existing knowledge graphs mentioned in section 4.2.3, the IMDB database <sup>2</sup> is chosen as a foundational database for the remaining experiments, due to its large size and easy to understand domain.

The schema provided by IMDB itself is not normalized and the data not assigned to readily distinguishable entities.

Therefore the database schema requires manual preprocessing before it can be used by the system further. The tables in the IMDB database schema were transformed into at least third normal form. Domain constraints were set, to guarantee unique values on non-id columns and make columns not nullable whenever applicable.

<sup>2</sup> See IMDB database schema at: <https://www.imdb.com/interfaces/>

The transformed relational model is depicted in figure 4.3. Two key entities *Person* and *Title* are identified. A title is any piece of media, such as a movie, television series, computer game, etc. A person is any human that is involved with a title in any way, such as directing or producing the title. Each person is assigned a name, a birth- and deathyear. A person can also have multiple professions, which in turn can be practised by multiple persons. A person can be associated to a title in multiple ways, such as directing it, while also acting in it.

Each title is assigned a name, format, an average rating with an amount of voters, a release year and whether it is rated adult. A title can have localized variants, that may have a different title name and an associated region and language. Each title may have multiple genres it belongs to, whereas a genre can be assigned to many titles.

#### 4.3.2 *Knowledge Graph Entities for Semantic Labeling of Tables*

To provide the additional metadata required by the SQL-query generation and SQL-query exercise description generation algorithms, a knowledge graph was manually crafted for the normalised IMDB database schema shown in figure 4.3. As the database already contains rows associated with the entities of the crafted relational database schema shown in figure 4.3, the instances of the knowledge graph class entities are omitted.

Thereby merely semantic labels for tables, table attributes and foreign key constraints are added, as well as imposing a directionality of semantic labels on foreign key constraints and junction tables. The cardinality between knowledge graph entities is additionally marked, to be able to model the need for junction tables in the physical database model.



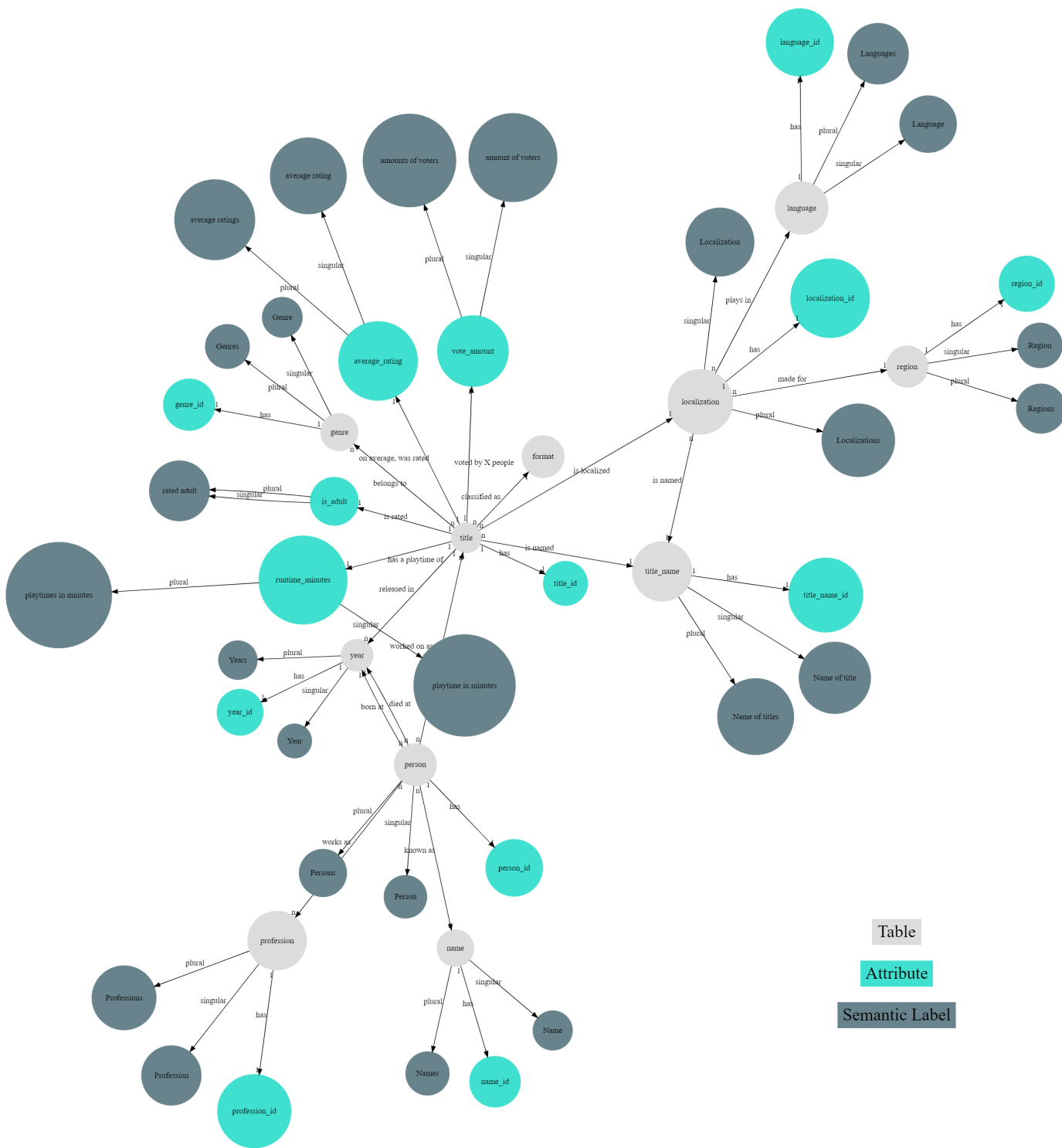


Figure 4.4: Handcrafted IMDB knowledge graph

Parameter	Type	Shape	Definition	Dependency
joinRange	Tuple<int,int>	$\{min..max\} \Rightarrow \{n \in \mathbb{N}^+ : min \leq n \leq max\}$	Determines the boundaries for the sampled amount of tables to use in the query. If more than one table is chosen, explicit JOIN statements are generated.	joinType
joinType	Array<string>	[ "INNER JOIN", "OUTER RIGHT JOIN", "OUTER LEFT JOIN", "FULL OUTER JOIN", "CROSS JOIN" ]	Determines the type of JOINS to sample from. Independent of the JOIN-type every JOIN is realised as an equi-join. The join-predicate is generated explicitly.	-
columnRange	Tuple<int,int>	$\{min..max\} \Rightarrow \{n \in \mathbb{N}^+ : min \leq n \leq max\}$	Determines the boundaries for the sampled amount of columns to use in the query.	-
constraintRange	Tuple<int,int>	$\{min..max\} \Rightarrow \{n \in \mathbb{N}^+ : min \leq n \leq max\}$	Determines the boundaries for the amount of constraints to use in the query.	constraintType
constraintType	Array<string>	[ "numericRange", "numericComparison", "nullComparison", "booleanEquality", "stringComparison", "stringFuzzyComparison" ]	Determines the type of constraints to sample from. - numericRange uses the BETWEEN keyword. - numericComparison uses the <>, <, >, = -operators - nullComparison uses the IS NULL keyword. - stringComparison uses the = operator for exact string matching - stringFuzzyComparison uses the = and % operators for fuzzy string matching	-
allowAggregates	Boolean	$\{false, true\}$	Determines whether aggregate functions are allowed or not. If aggregate functions are allowed, a random number of aggregate functions are sampled and applied to a random number of previously sampled columns. This also forces the generation of a GROUP BY clause.	aggregateType
aggregateType	Array<string>	[ "AVG", "COUNT", "MAX", "MIN", "SUM" ]	Determines the type of aggregate functions to sample from.	-
forceHavingClause	Boolean	$\{false, true\}$	Determines whether to generate a HAVING clause or not.	allowAggregates, aggregateType
forceOrderBy	Boolean	$\{false, true\}$	Determines whether to generate a ORDER BY clause or not. If a ORDER BY clause is generated, a random amount of columns are sampled and a random sort direction is chosen for each column.	-
schema	string	"imdb"	Determines the database to use for the query generation. Currently only supports the imdb-database.	-
seed	string	$s \in U^*$ where $ s  > 0$ <sup>3</sup>	If an arbitrary seed string is provided, the generation algorithm becomes deterministic and the result reproducible.	-

Table 4.1: Query generation algorithm parameters

## 4.4 GENERATION OF MEANINGFUL SQL-QUERY EXERCISES

## 4.4.1 Parameter Space of the Generation Algorithm

To satisfy the three criteria defined in section 3.1.1, the SQL-query generation algorithm receives the set of parameters defined in table 4.1.

## 4.4.2 Traversing the Relational Schema as a Graph of Tables and Foreign Key Constraints

---

**Algorithm 3** Table selection by generating a subgraph of the database schema

---

**Input:**  $R, T$

**Output:**  $S$

```

1:  $n \leftarrow \text{RANDOMINTEGERINRANGE}(R)$ 
2:  $t \leftarrow \text{SAMPLEWITHOUTREPLACEMENT}(T)$ 
3:  $\hat{T} \leftarrow t$ 
4:  $\hat{K} \leftarrow \text{empty set}$ 
5:  $K \leftarrow t.\text{edges}$ 
6: while  $n \neq 0 \mid K.\text{length} > 0$  do
7:    $k \leftarrow \text{SAMPLEWITHOUTREPLACEMENT}(K)$ 
8:   if  $\text{ISTARGETJUNCTIONTABLE}(k)$  and  $n > 2$  then
9:      $\hat{K}.\text{push}(k)$ 
10:     $\hat{T}.\text{push}(k.\text{target})$ 
11:     $k_2 \leftarrow \text{SAMPLEWITHOUTREPLACEMENT}(k.\text{target}.\text{edges})$ 
12:     $K.\text{push}(k.\text{target}.\text{edges} - k_2)$ 
13:     $\hat{T}.\text{push}(k_2.\text{target})$ 
14:     $K.\text{push}(k_2.\text{target}.\text{edges})$ 
15:     $n \leftarrow n - 2$ 
16:   else if not  $\text{ISNOTTARGETJUNCTIONTABLE}(k)$  then
17:      $K.\text{push}(k.\text{target}.\text{edges})$ 
18:      $\hat{K}.\text{push}(k)$ 
19:      $\hat{T}.\text{push}(k.\text{target})$ 
20:      $n \leftarrow n - 1$ 
21:   end if
22: end while
23:  $\hat{S} \leftarrow (\hat{T}, \hat{K})$ 
24: return  $\hat{S}$ 

```

---

The table selection algorithm shown in 3 receives a set of tables  $T$ .  $T$  and their foreign keys  $K$  are extracted from a given PostgreSQL database via the respective reflection queries listed in appendix A.2.

The parameter *joinRange*, defined in table 4.1, governs the amount of edges in the generated subgraph. A random, non-junction table  $t$  is sampled from  $T$ .

---

<sup>3</sup> Let  $U$  be the Unicode alphabet and  $U^*$  the set of all strings of any length over the alphabet  $U$ .

While  $n$  is not zero or  $K$  is not empty, sample an edge  $k$  from  $K$ . If the target of  $k$  is a junction table, two hops instead of one is made. This is done to generate more semantically plausible SQL-queries, as the junction tables represent a semantic relationship between two entities. Any edge, that was not already traveled is added to  $K$ , all traveled edges added to  $\hat{K}$  and all traveled nodes to  $\hat{T}$ .  $n$  is decreased by two.

If the target of  $k$  is not a junction table, only add the unseen edges of  $k.target$  to  $K$ , the travelled edge  $k$  to  $\hat{K}$  and  $k.target$  to  $T$ .  $n$  is decreased by one.

The set of tables  $T$  and the set of foreign keys  $K$  are transformed into an undirected graph  $S = (T, K)$  and returned.

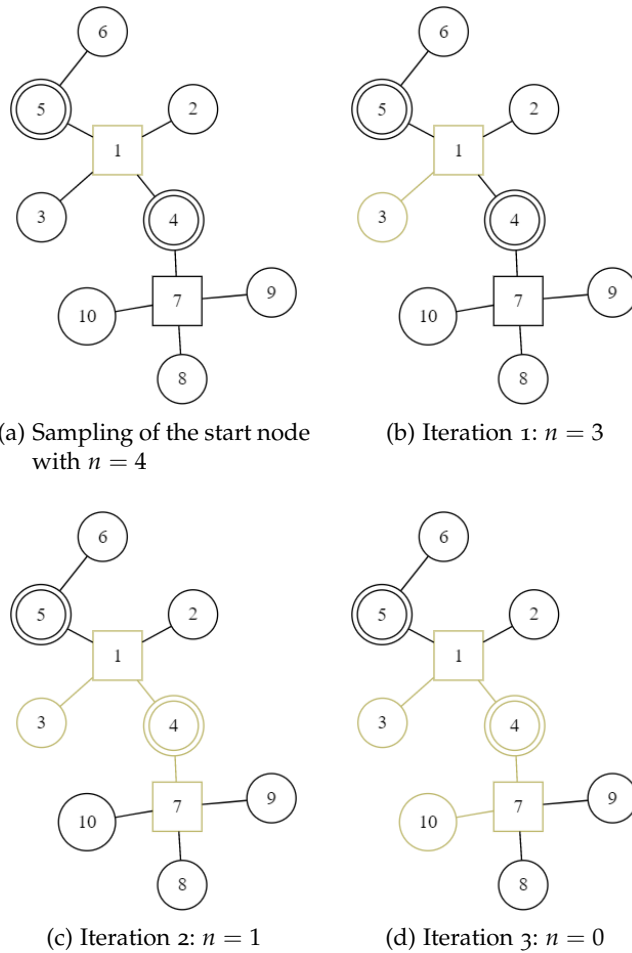


Figure 4.5: Schema path selection example with  $n = 4$  hops

A step by step example of the algorithm 3 is visualized in figure 4.5.

#### 4.4.3 Generation of Random Parameter-Compliant SQL-Queries

Once the table selection is performed by generating the JOIN path with the algorithm outlined in 3, the remaining SQL-query constituents are generated in the order defined in listing 4.1.

```
[WHERE CONSTRAINTS]
SELECT COLUMN
[GROUP BY]
[HAVING CONSTRAINTS]
[ORDER BY]
```

Listing 4.1: SQL-query generation order

```
NOT BETWEEN, BETWEEN, <>, <, >, <=, >=, =
```

Listing 4.2: Numeric constraint operators

```
LIKE, NOT LIKE, <>, =
```

Listing 4.3: String constraint operators

The WHERE-clause allows for four sets of operations, depending on the datatype and whether the selected column is nullable. The supported constraint sets are 1.) numeric, 2.) string, 3.) boolean 4.) null. Numeric operands can be constrained by the operators in listing 4.2. String operands can be constrained by the operators in listing 4.2. Boolean operands can be constrained by the =-operator and either the value *true* or *false*. To generate the constraints, a number  $n$  in the range specified by the *constraintRange*-parameter is sampled. Then a set of uniquely named columns  $C$ , across the previously selected tables is constructed. The set  $C$  is then filtered by the supported datatypes of the selected constraint types in the *constraintType*-parameter to create the set  $C_f$ .  $n$  columns are sampled from  $C_f$  and for every  $c$  therein a random constraint type, that adheres to the domain constraint of the column, is then sampled. If multiple constraints are generated, they are concatenated randomly with either the *AND* or *OR* keyword.

For the *numericRange*-constraint, the algorithm randomly chooses between the operator *BETWEEN* or *NOT BETWEEN* and samples two random distinct values of the column data as operands.

For the *numericComparison*-constraint, the algorithm randomly chooses a relational operator and samples a random value of the column data. To avoid empty return sets, the min and max value of the column data are not eligible as operands.

For the *nullComparison*-constraint, the algorithm randomly chooses between the operator *IS NULL* or *IS NOT NULL*.

For the *stringComparison*-constraint, the algorithm randomly chooses between applying the *<>* or = operator and samples a random value of the column data.

For the *stringFuzzyComparison*-constraint, the algorithm randomly chooses between applying the *LIKE* or not *NOT LIKE* operator and samples a random

value of the column data. The operand is then sliced into a substring by selecting random indices. This may also return the original substring if the indices match the boundaries of the operand. %-signs are appended to the left most outer and right most outer position of the operand, if any character was sliced off either end.

The SELECT-clause is generated by choosing a random number  $i$  in the range specified by the *columnRange* and again sampling  $i$  random columns of the set of uniquely named columns  $C$  of all selected tables. If the *allowAggregates*-parameter is *true*, a random number of numeric, non-id columns is selected and a randomly chosen aggregate function specified in the *aggregateType*-parameter is applied to the column. If an aggregate function was applied, a GROUP BY-clause is generated, by applying every non-aggregate column as operands.

If the *forceHavingClause*-parameter is set to *true*, an aggregate function and GROUP BY-clause is generated. The HAVING-clause then chooses a random amount of aggregated columns and applies either a *numericComparison* or *numericRange* constraint to the chosen operands.

If the *forceOrderBy*-parameter is set to *true*, an ORDER BY-clause is generated. The operands are randomly sampled from the set of uniquely named columns  $C$  of all selected tables and a random sorting order of the options *ASC* or *DESC* is applied.

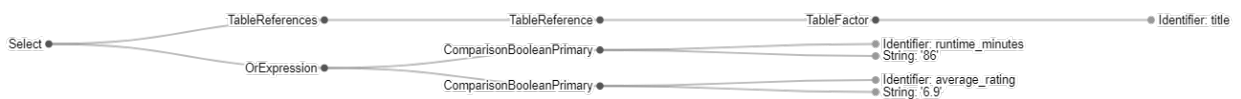


Figure 4.6: SQL-query AST example

The generated constituents are then transformed into a AST-representation of the SQL-query, as shown in example 4.6.

```

SELECT year, is_adult
FROM title as t
WHERE runtime_minutes < '86' OR average_rating >= '6.9';
  
```

Listing 4.4: SQL-query string example

The AST-representation is then traversed to generate the textual SQL-query string, as shown in example 4.4.

## 4.5 GENERATION OF AN SQL-QUERY EXERCISE FORMULATION IN NATURAL LANGUAGE

### 4.5.1 Macroplanning the SQL-Query Exercise Formulation

```

COLUMN TABLE|JOIN [WHERE CONSTRAINTS] [GROUP BY] [HAVING] [ORDER BY]
  
```

Listing 4.5: SQL-query description structuring

The generation of the SQL-query exercise description follows the pipeline outlined in section 2.2.2.

As the content of the SQL-query can already be controlled by the parametrization of the SQL-query generation algorithm, no additional content determination is required. The input of the NLG-pipeline is a SQL-query-AST produced by the SQL-query generation algorithm outlined in the previous section 4.4.

The NLG-pipeline functions in two modes: **A1**, a rudimentary rule-based templating approach as a baseline, and **A2**, a hybrid approach that utilizes a Large Language Model (LLM). The differences between the approaches in the NLG-pipeline are mentioned as they occur in the following.

The structuring of the exercise description follows the same predefined pattern for both **A1** and **A2**. Since SQL is modeled closely to read like the English language, the pattern is outlined in the general order of keyword specification, as shown in listing 4.5.

#### 4.5.2 *Microplanning the SQL-Query Exercise Formulation*

The lexicalization component of the NLG-pipeline utilizes two repositories to select and assign the appropriate expressions to the generated SQL-query-constituents and operands. Repository a): The handcrafted semantic node labels and the semantic descriptors assigned to the edges, which are defined in the knowledge graph displayed in figure 4.4 and repository b): An assortment of generic templates that directly translate the semantics of a particular SQL-query keyword or operator.

**A2** utilizes the semantic labels of repository a) to assign natural sounding table and attribute names, as well as specifying the semantic label of eventual relationships between entities. The templates provided by repository b) are utilized by both **A1** and **A2**, to formulate the operational semantics of the SQL symbols. **A2** also utilizes placeholders in the shape of *[MASK]*, that are replaced at a later stage.

The template repositories for both approaches are attached in appendix A.3.

**A2** also utilizes aggregation and referring expression generating components, to a) reduce redundant entity mentions, b) shorten the text output and c) hide unnecessary technical details by employing implicit phrasing.

#### 4.5.3 *Surface Realization of the Structured Exercise Formulation*

**A2** utilizes a LLM as a linguistic realization component. The previously introduced *[MASK]* tokens are unmasked by the DistilBERT model [59]. The mask filling approach gives the sentence structure a less rigid feel. On the negative side, the output is not controllable, so there is a chance of unfitting words being introduced.

Neither **A1**, nor **A2** make use of a structure realization component.

Find the release year and whether the titles are rated adult or not for all titles, for which the runtime is smaller than '86' and the average rating is greater or equal to '6.9'.

(a) Hybrid NLG-pipeline, highlighted text was previously masked

Use table title. Return the columns release\_date\_id and is\_adult. Only return the data for which runtime\_minutes is smaller than '86', average\_rating is greater or equal than '6.9' is true.

(b) Base NLG-pipeline

Figure 4.7: SQL-query example in 4.4 of baseline vs. hybrid NLG-pipeline

As figure 4.7 shows, there are only minor differences between the two NLG-pipeline approaches in terms of length for the translation of simple queries. That said, by providing semantic labels for column names, the description reads less technical and utilizing mask filling allows for embedded clauses.

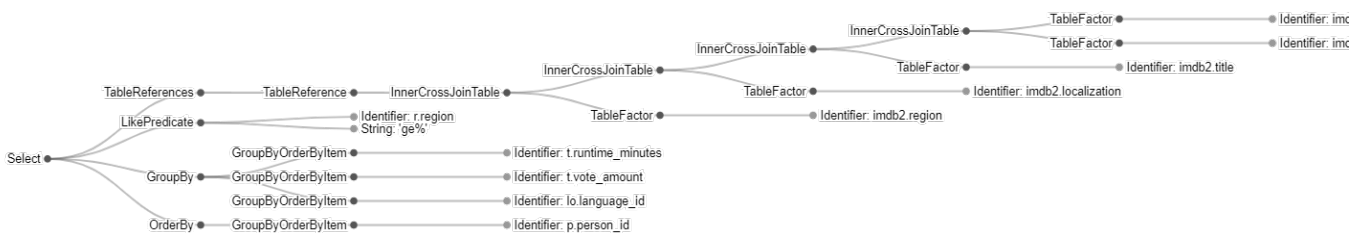


Figure 4.8: Complex SQL-query AST

```
SELECT COUNT(t.title_id), t.runtime_minutes, t.vote_amount, lo.
  language_id
FROM imdb2.person as p
INNER JOIN imdb2.year as y
ON p.birthyear_id = y.year_id
INNER JOIN imdb2.title as t
ON y.year_id = t.release_date_id
INNER JOIN imdb2.localization as lo
ON t.title_id = lo.title_i
INNER JOIN imdb2.region as r
ON lo.region_id = r.region_id
WHERE r.region LIKE 'ge%'
GROUP BY t.runtime_minutes, t.vote_amount, lo.language_id
HAVING COUNT(t.title_id) <> '4883174'
ORDER BY p.person_id ASC;
```

Listing 4.6: Complex SQL-query string example

When the two approaches are compared on more complex queries, the differences become clearer, as shown in 4.9, for the complex query displayed in 4.6.



Find the count of title-ids, the runtime, the number of votes and the language-ids for all persons and the titles they were involved with, if the region contains 'ge' and the count of title\_id is not '4883174'. Sort the result by the person-id in ascending order.

(a) Hybrid NLG-pipeline

Form the intersection that contains the corresponding entries of the tables person and year and the intersection that contains the corresponding entries of the tables year and title and the intersection that contains the corresponding entries of the tables title and localization and the intersection that contains the corresponding entries of the tables localization and region. Return the columns the amount of title\_id, runtime\_minutes, vote\_amount and language\_id. Only return the data for which region contains 'ge'. A further constraint is the amount of title\_id doesn't equal '4883174'. Group the result by runtime\_minutes, vote\_amount, title\_id, language\_id and person\_id. Sort the result ascending by person\_id.

(b) Base NLG-pipeline

Figure 4.9: Complex SQL-query example of baseline vs. hybrid NLG-pipeline

## EVALUATION OF THE SYSTEM FOR THE GENERATION OF MEANINGFUL SQL-QUERY EXERCISES

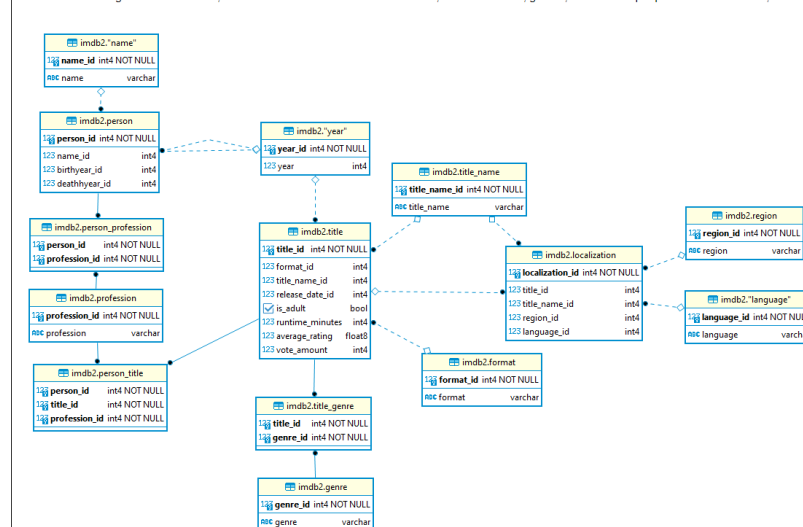
### 5.1 EVALUATION STUDY DESIGN

#### 5.1.1 Crowdsourcing for Evaluating the Generated Exercise Component

In order to evaluate the system for generating SQL-query exercises the crowdsourcing framework Amazon Mechanical Turk <sup>1</sup> was used. Each assessment was rated by three judges. The raters were selected, if they had self-proclaimed knowledge of SQL.

A total number of 667 assessments of individual SQL-queries were evaluated. The payment per query evaluation was 0.40\$ with an estimated evaluation time of 2-3 minutes, yielding a hourly rate of 8 to 12\$.

Given is the following database schema, that describes a database for media titles, such as movies, games, etc. and the people that work on them, such as actors, directors, writers, etc.



The following SQL-query and the corresponding result (first 10 rows) are based on the above database schema.

Figure 5.1: Assessment opening screen

The assessment were all structured equally. A introductory task description alongside the database schema open each assessment, as seen in screenshot 5.1

As seen on screenshot 5.2, every assessment provides a SQL-query screenshot and the matching first 10 rows of the resultset. The rater is then asked to rate how likely it is, that the information in the presented SQL-query is requested by a human for the given database on a 5 point likert scale. Also on a 5 point likert scale the rater is asked to judge how frequently the infor-

<sup>1</sup> <https://www.mturk.com/>

```

1 SELECT p."deathyear_id", p."person_id", t."format_id"
2 FROM imdb2."person" as p
3 INNER JOIN imdb2."year" as y
4 ON p."birthyear_id" = y."year_id"
5 INNER JOIN imdb2."title" as t
6 ON y."year_id" = t."release date id"
7 WHERE y."year_id" > '305' AND t."title_id" < '141866'
8 ORDER BY y."year" DESC;

```

	deathyear_id	person_id	format_id
0	NaN	11017435	7
1	340.0	5534958	7
2	NaN	9803099	7
3	NaN	11017435	7
4	340.0	5534958	7
5	NaN	9803099	7
6	NaN	11017435	7
7	340.0	5534958	7
8	NaN	9803099	7
9	NaN	10067598	7

Given this SQL-query and database schema, rate the query properties and evaluate the english descriptions of the SQL-query underneath.

**How plausible is it for a human to request the information expressed by this SQL-query for this database?**

Very likely    Likely    Context dependent    Unlikely    Very unlikely

**How often would you guess is the information expressed by this SQL-query requested?**

Very frequent    Frequent    Context dependent    Rarely    Very rarely

**On a scale of 1-10, how complex would you rate this query?**

Figure 5.2: General questions about the SQL-query

information expressed by the SQL-query is requested. On a numeric scale of 1 to 10, the rater is then asked to rate the complexity of the given SQL-query.

Instructions
Shortcuts
Select the error categories that apply to the text

Form the intersection that contains the corresponding entries of the tables person and year and the intersection that contains the corresponding entries of the tables year and title. Return the columns deathyear\_id, person\_id and format\_id. Only return the data for which year\_id is greater than '305', title\_id is smaller than '141866' is true. Sort the result descending by year.

Select appropriate categories

Positive	1
Negative	2
Akward style	3
Missing information	4
Incomprehensible	5
None of the above	n

Figure 5.3: Error categories per NLG-approach

Screenshot ?? requests the raters to judge eventually applicable error categories per natural language description of each NLG-approach.

Lastly, the raters are asked which of the two presented natural language descriptions of the SQL-query were deemed more human-like.

Which of the english descriptions would you describe as more human-like?

1    2

Figure 5.4: Human-likeness of the SQL-query descriptions

## 5.2 EVALUATION OF THE EXERCISE GENERATION ALGORITHM FEATURES

### 5.2.1 *Evaluation of the Generation of an Exercise Formulation in Natural Language*

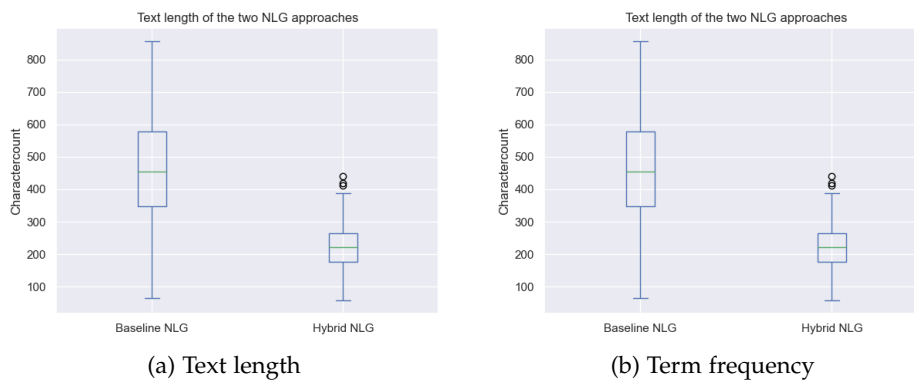
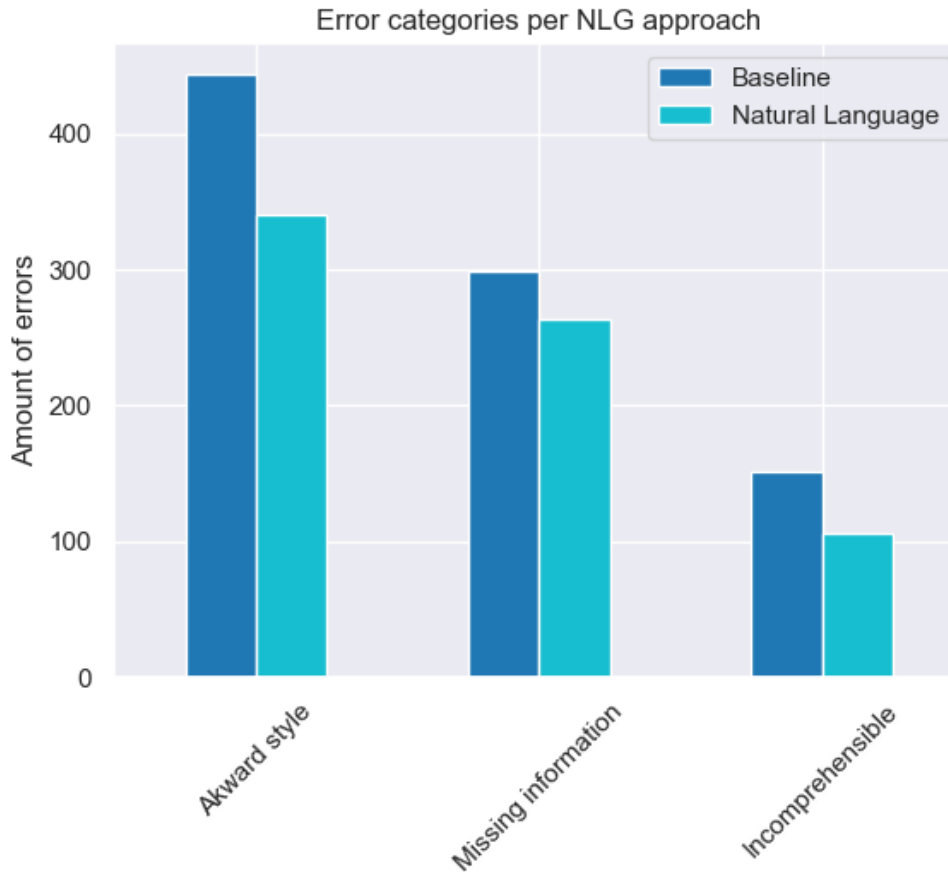


Figure 5.5: Comparison of Exercise Formulation in Natural Language

As figure 5.5a and figure 5.5b show, compared to the baseline, the hybrid NLG-approach generates natural language descriptions of SQL-queries, which are on average 47.87% shorter and exhibit a  $\sim 20\%$  decrease in term frequency. Also, by applying aggregation and referring expression generating techniques and using semantic labels for foreign key constraints, the hybrid approach minimizes explicit mentions of database normalisation induced technicalities.

Thus the hybrid approach is deemed more concise, by the criteria defined in 3.1.2.

The raters assigned each error category more often to the natural language description generated by the baseline approach. The difference in correctness is therefore considerably smaller, although the hybrid approach still outperforms the baseline. The largest margin is gained on the missing information error category, which is interesting as the hybrid approach is on average only half as long. This suggests that really only redundant and implicitly clear information has been removed.



### 5.2.2 Evaluation of the Automatic Assessment of the Solution Attempts

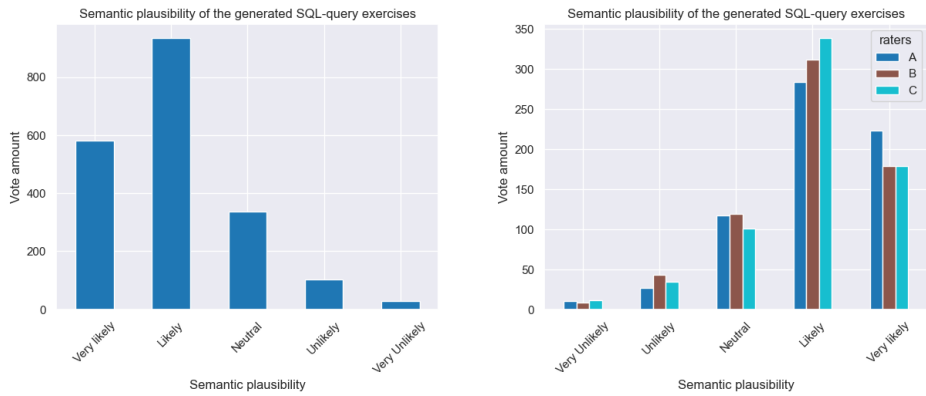
The SQL-query exercise generation system is able to decide on the correctness of a solution attempt by matching the result sets of the user query and the generated query. The generation of individual scores and hints was not fulfilled, therefore this requirement is viewed as only partially satisfied.

## 5.3 EVALUATION OF THE CONSISTENCY AND QUALITY OF THE GENERATED SQL-QUERY EXERCISES

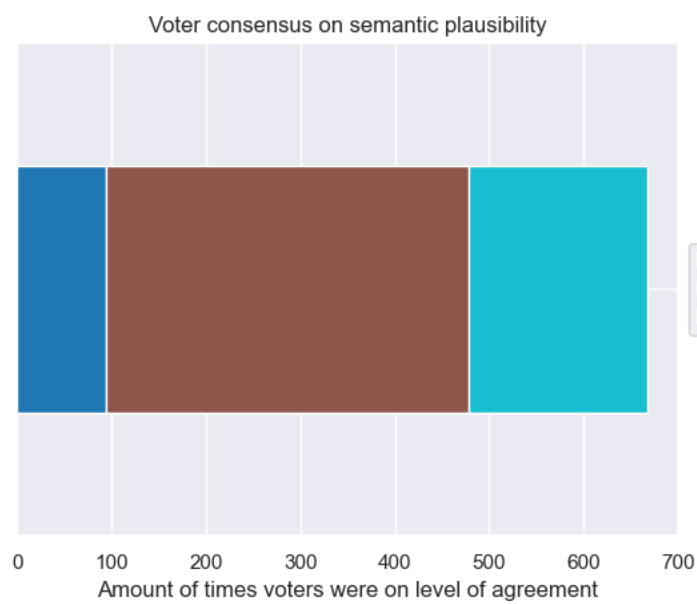
### 5.3.1 Evaluation of the Syntactic Soundness of the Generated Exercise

The generation of syntactically sound SQL-queries was achieved, as all generated queries can be executed and follow the restrictions defined in section 3.2.1, such as always producing the SELECT and FROM keyword and selecting all non-aggregate columns to be specified as grouping columns.

### 5.3.2 Evaluation of the Semantic Plausibility of the Generated Exercise



(a) Global rated semantic plausibility of generated SQL-query (b) Rated semantic plausibility of generated SQL-query per voter



(c) Voter consensus

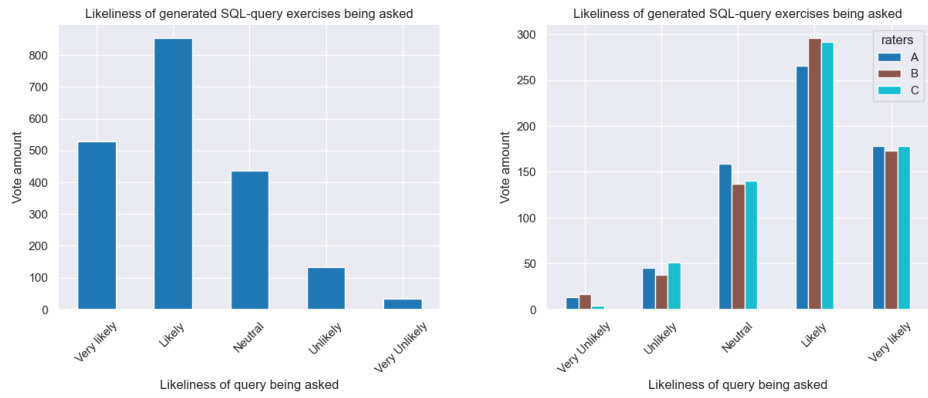
Figure 5.6: Comparison of how semantically plausible the raters judge a generated query to be

## 5.4 EVALUATION OF THE SQL-QUERY EXERCISE GENERATION SYSTEM PROPERTIES

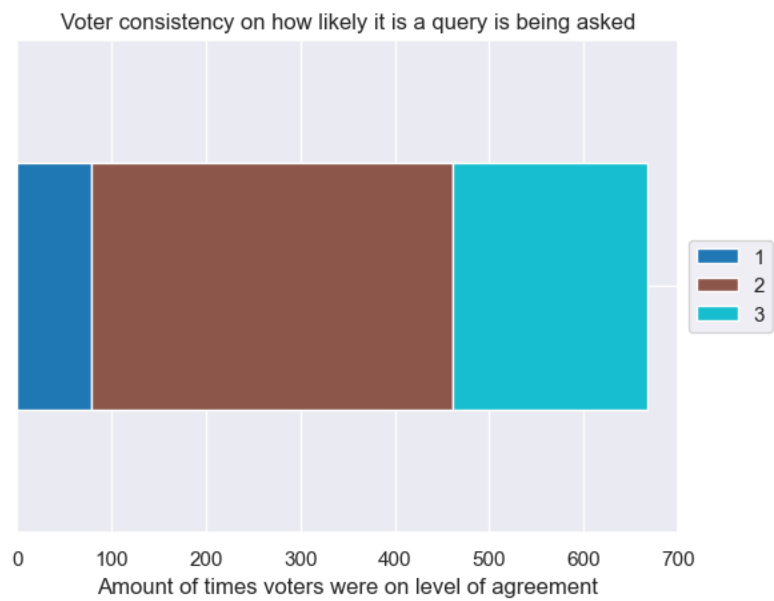
### 5.4.1 Evaluation of the Performance of the Exercise Generation System

As seen in figure ?? the query exercise generation system generates the exercises in a reasonable amount of time. Generally well below a second.

As seen in ??, the generation of some queries require longer amounts of time with up to a minute. This is due to those generated queries, requiring to retrieve information from the database often during the generation process. This is especially expensive when generating queries including a HAVING-



(a) Global likeliness of generated SQL-query being asked (b) Likelihood of generated SQL-query being asked per voter



(c) Voter consensus

Figure 5.7: Comparison of how likely raters judge a generated query to be asked

clause, as sensible values need to be extracted by querying the database with the, by then, almost fully generated SQL-query.

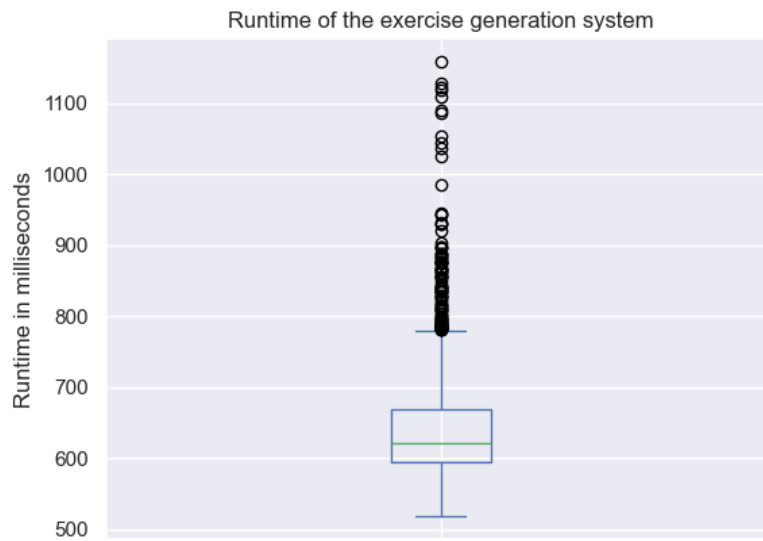


Figure 5.8: Runtime of generating a SQL-query exercise without outliers

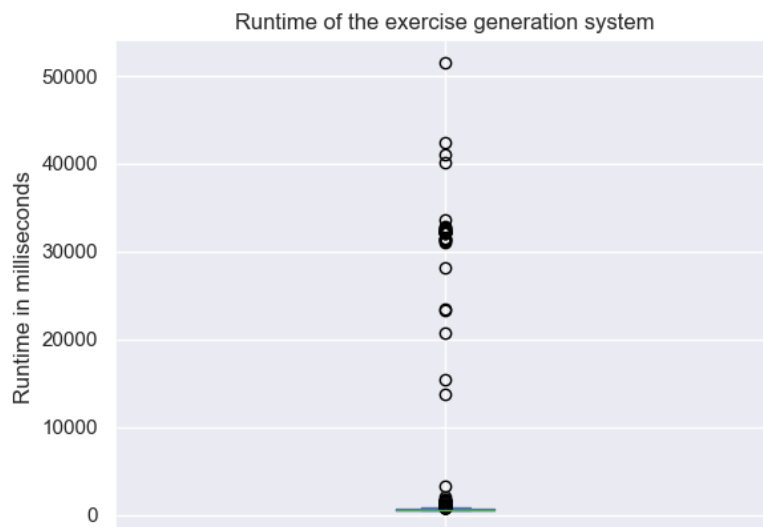


Figure 5.9: Runtime of generating a SQL-query exercise with outliers



## CONCLUSION AND FUTURE WORK

---

### 6.1 CONCLUSION

A system for the generation of SQL-query exercises was developed and presented. It features a SQL-query generation algorithm that allows for selection of necessary SQL-constituents that are required to be generated in the SQL-query. The generated query-exercises are syntactically correct and are deemed semantically plausible. The SQL-query generation also includes more keywords than previous approaches and thus allows for the generation of more varied exercises. The system is also capable of determining the correctness of any solution attempt by matching the query result sets.

The system also provides a SQL-query exercise description in natural language, which is generated based on the SQL-query. The natural language description is evaluated against a baseline approach which it outperforms by a large margin on the metrics defined in 3.1.2. Still, it is found that the generated natural language descriptions are often to be found written in awkward style, in less cases even incomprehensible and exhibited missing information according to the evaluation. It was also found that in order to generate high quality SQL-query exercises, some manual labor in creating an appropriate metamodel from which to generate the SQL-queries or the relational databases from is required. This is due to lacking data quality in existing resources from which the required task artefacts could be generated automatically, or semi-automatically.

### 6.2 FUTURE WORK

Future work will include the testing of the system for generating meaningful SQL-query exercises in educational settings to determine whether students benefit from the proposed approach. Due to the not fully satisfying natural language descriptions of the generated SQL-queries, more research has to be done on how to write them in a more human-like fashion, without compromising correctness and comprehensiveness. An interesting approach could be the use of composition methods in natural language embeddings [27, 52] to mirror SQL operations in the latent space of the embeddings, such as applying different filter constraints in WHERE or HAVING clauses. For example, a numeric constraint that restricts an column called *age* of a table called *person*, to only contain values lower than 10 changes the semantic identifier of the table to, e.g. *minor* or *child*. Although how these operations are supposed to be mirrored in the embedding space is still unclear. Also, new measures on cognitive complexity in regards to the writing of SQL-queries and writing programs in general could be of use, as current metrics

do not take into consideration the different weighting of individual constituents and the complexity arising out of eventual synergies that arise out of specific constellations of query-constituents. Additionally, the approach to generate [DQL](#)-query exercises is envisaged to be expanded to different, larger subsets of the SQL-language.

Part II

APPENDIX



## APPENDIX

---

### A.1 DQL KEYWORD SUBSET

```
SELECT <select_list>
  FROM <table_reference_list>
  [ <where_clause> ]
  [ <group_by_clause> ]
  [ <having_clause> ]
  [ <order_by_clause> ]

<select_list> ::=
  <column_specifier> [ { , <column_specifier> } ... ]
  | [ <aggregate_function>(<column_specifier> ) ]

<column_specifier> ::=
  [table_alias.] <column_specifier>

<aggregate_function> ::=
  AVG
  | MAX
  | MIN
  | COUNT
  | SUM

<table_reference_list> ::=
  [ schema_name. ] table_name [ AS table_alias ]
  | <joined_table>

<joined_table> ::=
  table_name [ <join_type> ] JOIN table_name <join_specification>

<join_type> ::=
  INNER
  | LEFT [OUTER]
  | RIGHT [OUTER]
  | FULL [OUTER]

<join_specification> ::=
  ON <predicate_clause>

<predicate_clause> ::=
  <predicate>
  [ <concatenation> <predicate> ]

<concatenation> ::=
  AND
```

```

| OR

<predicate_clause> ::=
    <comparison_predicate>
    | <between_predicate>
    | <like_predicate>
    | <null_predicate>
    | <boolean_predicate>

<comparison_predicate> ::=
    <column_specifier> <comp_op> column_value

<comp_op> ::=
    =
    | <>
    | <
    | >
    | <=
    | >=

<between_predicate> ::=
    <column_specifier> [ NOT ] BETWEEN column_value AND column_value

<like_predicate> ::=
    <column_specifier> [ NOT ] LIKE [ % ] string [ % ]

<null_predicate> ::=
    <column_specifier> IS [ NOT ] NULL

<boolean_predicate> ::=
    <column_specifier> IS [ NOT ] <truth_value>

<truth_value> ::=
    TRUE
    | FALSE

<where_clause> ::=
    WHERE <predicate_clause>

<group_by_clause> ::=
    GROUP BY <column_specifier> [ { , <column_specifier> } ... ]

<having_clause> ::=
    HAVING <predicate_clause>

<order_by_clause> ::=
    ORDER BY <column_specifier> <sort_direction>

<sort_direction> ::=
    ASC
    | DESC

```

Listing A.1: DQL keyword subset

## A.2 POSTGRESQL REFLECTION QUERIES

## A.2.1 PostgreSQL Table Reflection Query

```

DECLARE @schema VARCHAR
SET @schema = 'imdb'

SELECT columns.table_name,
       columns.column_name,
       columns.data_type
FROM information_schema.columns
WHERE table_name in
      (SELECT tables.table_name
       FROM information_schema.tables
       WHERE tables.table_schema = @schema
         AND tables.table_name != 'schema_version'
         AND tables.table_type = 'BASE TABLE');

```

Listing A.2: Table reflection query

## A.2.2 PostgreSQL Foreign Key Reflection Query

```

DECLARE @schema VARCHAR
SET @schema = 'imdb'

SELECT m.relname AS source_table,
      (SELECT a.attname FROM pg_attribute a
       WHERE a.attrelid = m.oid
         AND a.attnum = o.conkey[1]
         AND a.attisdropped = false)
      AS source_column,
      f.relname AS target_table,
      (SELECT a.attname
       FROM pg_attribute a
       WHERE a.attrelid = f.oid
         AND a.attnum = o.confkey[1]
         AND a.attisdropped = false)
      AS target_column
FROM pg_constraint o
  LEFT JOIN pg_class f ON f.oid = o.confrelid
  LEFT JOIN pg_class m ON m.oid = o.conrelid
WHERE o.contype = 'f'
  AND o.conrelid IN (SELECT oid FROM pg_class c WHERE c.relkind = 'r')
  AND o.connamespace::regnamespace::text = @schema';

```

Listing A.3: Foreign Key reflection query

## A.3 NLG TEMPLATE REPOSITORIES

A.3.1 *Baseline NLG Templates*

```

conjunctions: {
  "&": " and ",
  "|": " or ",
},
joinTemplate: {
  nonJoinStartingPhrase: "Use table ${table}.",
  joinStartingPhrase: "Form",
  joinTypes: {
    "RIGHT OUTER JOIN": "the intersection that contains all
      entries of ${source} and the corresponding entries
      of ${target}",
    "LEFT OUTER JOIN": "the intersection that contains all
      entries of ${target} and the corresponding entries
      of ${source}",
    "CROSS JOIN": "the cross product of the tables ${source}
      and ${target}",
    "INNER JOIN": "the intersection that contains the
      corresponding entries of the tables ${source} and ${
      target}",
  },
},
booleanTemplate: "",
aggregationTemplate: {
  SUM: "the sum of ${column}",
  AVG: "the average of ${column}",
  MAX: "the maximum of ${column}",
  MIN: "the minimum of ${column}",
  COUNT: "the amount of ${column}",
},
operatorTemplate: {
  BETWEEN: "is between ${value1} and ${value2}",
  "<": "doesn't equal",
  "<": "is smaller than",
  ">": "is greater than",
  "<=": "is smaller or equal than",
  ">=": "is greater or equal than",
  "=": "equals",
  LIKE: {
    0: "contains '${value}'",
    1: "ends with '${value}'",
    2: "starts with '${value}'",
  },
  "NOT LIKE": {
    0: "doesn't contain '${value}'",
    1: "doesn't end with '${value}'",
    2: "doesn't start with '${value}'",
  },
},
NULL: "NULL",

```

```

    "NOT NULL": "not NULL",
  },
  columnTemplate: {
    columnStartingPhrasePlural: "Return the columns",
    columnStartingPhraseSingular: "Return the column",
    columnEndingPhrase: "",
  },
  constraintTemplate: {
    startingPhrase: "Only return the data for which",
    endingPhrase: "is true",
    LIKEOperatorFallback: { exclude: "contains any string",
      include: "contains an empty string" },
  },
  groupByTemplate: {
    startingPhrase: "Group the result by",
    endingPhrase: "",
  },
  havingTemplate: {
    startingPhrase: "A further constraint is",
    endingPhrase: "",
  },
  orderByTemplate: {
    startingPhrase: "Sort the result",
    endingPhrase: "",
    direction: {
      ASC: "ascending",
      DESC: "descending",
    },
  },
}

```

Listing A.4: Baseline SQL-constituent templates

### A.3.2 Hybrid NLG templates

```

conjunctions: {
  "&": " and ",
  "|": " or ",
},
joinTemplate: {
  nonJoinStartingPhrase: "",
  joinStartingPhrase: "",
  joinTypes: {
    "RIGHT OUTER JOIN": "${target} and [MASK] ${source},
      even if [MASK] have no ${target}",
    "LEFT OUTER JOIN": "${source} and [MASK] ${target}, even
      if [MASK] have no ${source}",
    "CROSS JOIN": "das kartesische Produkt der Tabellen ${
      source} und ${target}",
    "INNER JOIN": "${source} and [MASK] corresponding ${
      target}",
  },
},

```



```

},
booleanTemplate: "wether the ${table} is ${column} or not",
aggregationTemplate: {
  SUM: "the sum of ${column}",
  AVG: "the average of ${column}",
  MAX: "the maximum of ${column}",
  MIN: "the minimum of ${column}",
  COUNT: "the count of ${column}",
},
operatorTemplate: {
  BETWEEN: "is between ${value1} and ${value2}",
  "<": "is not",
  "<": "is smaller than",
  ">": "is larger than",
  "<=": "is smaller or the same as",
  ">=": "is greater or the same as",
  "=": "is",
  LIKE: {
    0: "contains '${value}'",
    1: "ends with '${value}'",
    2: "starts with '${value}'",
  },
  "NOT LIKE": {
    0: "doesn't contain '${value}'",
    1: "doesn't end with '${value}'",
    2: "doesn't start with '${value}'",
  },
  NULL: "NULL",
  "NOT NULL": "not NULL",
},
columnTemplate: {
  columnStartingPhrasePlural: "",
  columnStartingPhraseSingular: "",
  columnEndingPhrase: "",
},
constraintTemplate: {
  startingPhrase: "",
  endingPhrase: "",
  LIKEOperatorFallback: { exclude: "contains any string",
    include: "contains an empty string" },
},
groupByTemplate: {
  startingPhrase: "Group the result by",
  endingPhrase: "",
},
havingTemplate: {
  startingPhrase: "",
  endingPhrase: "",
},
orderByTemplate: {
  startingPhrase: "Sort the result by ",
  endingPhrase: "",
}

```

```
direction: {  
  ASC: "in ascending order",  
  DESC: "in descending order",  
},  
}
```

Listing A.5: Hybrid SQL-constituent templates

## BIBLIOGRAPHY

---

- [1] David Abián, Francesco Guerra, J. Martínez-Romanos, and Raquel Trillo Lado. "Wikidata and DBpedia: A Comparative Study." In: *International KEYSTONE Conference*. 2017.
- [2] Alireza Ahadi, Vahid Behbood, Arto Vihavainen, Julia Coleman Prior, and Raymond Lister. "Students' Syntactic Mistakes in Writing Seven Different Types of SQL Queries and its Application to Predicting Students' Success." In: *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (2016).
- [3] Alireza Ahadi, Julia Coleman Prior, Vahid Behbood, and Raymond Lister. "A Quantitative Study of the Relative Difficulty for Novices of Writing Seven Different Types of SQL Queries." In: *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (2015).
- [4] Kanokwan Atchariyachanvanich, Srinual Nalintippayawong, and Thanakrit Julavanich. "Reverse SQL Question Generation Algorithm in the DBLearn Adaptive E-Learning System." In: *IEEE Access* 7 (2019), pp. 54993–55004.
- [5] "Automatic Construction of Domain-Specific Knowledge Graphs from Wikidata." In: 2020.
- [6] Adrien Basse, Baboucar Diatta, and Samuel Ouya. "Ontology-Based System for Automatic SQL Exercises Generation." In: *IMCL*. 2019.
- [7] Maria Becker, Siting Liang, and Anette Frank. "Reconstructing Implicit Knowledge with Language Models." In: *Workshop on Knowledge Extraction and Integration for Deep Learning Architectures; Deep Learning Inside Out*. 2021.
- [8] Armand Bosch. "WikiDataSets : Standardized sub-graphs from Wiki-Data." In: *ArXiv abs/1906.04536* (2019).
- [9] Hans Chalupsky, Pedro A. Szekely, Filip Ilievski, Daniel Garijo, and Kartik Shenoy. "Creating and Querying Personalized Versions of Wikidata on a Laptop." In: *ArXiv abs/2108.07119* (2021).
- [10] Mr. Sumit Chaudhari, Mr. Aditya Hire, Ms. Bhagyashree Mandale, and Ms. Sharayu Vanjari. "Structural Query Language Question Creation by using Inverse Way." In: 2021.
- [11] Naihao Deng, Yulong Chen, and Yue Zhang. "Recent Advances in Text-to-SQL: A Survey of What We Have and What We Expect." In: *COLING*. 2022.
- [12] Quan Chau Dong Do, Rajeev Agrawal, Dhana Rao, and Venkat N. Gudivada. "Automatic Generation of SQL Queries." In: 2014.

- [13] Abhilasha A. Dwivedi and Dinesh D. Patil. "AUTOMATIC SQL QUESTION GENERATION USING REVERSE APPROACH." In: 2020.
- [14] Lisa Ehrlinger and Wolfram Wöß. "Towards a Definition of Knowledge Graphs." In: *International Conference on Semantic Systems*. 2016.
- [15] Stavroula Eleftherakis, Orest Gkini, and Georgia Koutrika. "Let the Database Talk Back: Natural Language Explanations for SQL." In: *SEA-DataVLDB*. 2021.
- [16] Juliette Faille, Albert Gatt, and Claire Gardent. "The Natural Language Generation Pipeline Neural Text Generation and Explainability." In: 2020.
- [17] Michael Färber. "Which Knowledge Graph Is Best for Me ? " Linked Data Quality of DBpedia , Freebase , OpenCyc , Wikidata , and YAGO " in a Nutshell." In: 2018.
- [18] Jose Emilio Labra Gayo. "Creating Knowledge Graphs Subsets using Shape Expressions." In: *ArXiv abs/2110.11709* (2021).
- [19] Jose Emilio Labra Gayo. "WShEx: A language to describe and validate Wikibase entities." In: *ArXiv abs/2208.02697* (2022).
- [20] Ryan Colin Gibson and Gordon Morison. "Improving Student Engagement and Active Learning with Embedded Automated Self-assessment Quizzes: Case Study in Computer System Architecture Design." In: *Lecture Notes in Networks and Systems* (2021).
- [21] Venkat N. Gudivada, Kamyar Arbabifard, and Dhana Rao. "Automated Generation of SQL Queries that Feature Specified SQL Constructs." In: 2017.
- [22] Ramanathan V. Guha, Dan Brickley, and Steve Macbeth. "Schema.org: Evolution of Structured Data on the Web." In: *Queue* 13 (2015), pp. 10–37.
- [23] Armin Haller and Axel Polleres. "Are we better off with just one ontology on the Web?" In: *Semantic Web* 11 (2020), pp. 87–99.
- [24] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. USA: Elsevier Science Inc., 1977. ISBN: 0444002057.
- [25] Nicolas Heist and Heiko Paulheim. "Uncovering the Semantics of Wikipedia Categories." In: *ArXiv abs/1906.12089* (2019).
- [26] Daniel Henselmann and A. Harth. "Constructing demand-driven Wikidata Subsets." In: *WikidataISWC*. 2021.
- [27] Dayananda Herurkar, Philipp Blandfort, Federico Raue, Jörn Hees, and Andreas R. Dengel. "ANP-W2V: Effects of Composition Methods for Embedding Adjective-Noun Pairs." In: *2021 International Joint Conference on Neural Networks (IJCNN)* (2021), pp. 1–8.
- [28] Seyed Amir Hosseini Beghaeiraveri, Alasdair Gray, and Fiona Mcneill. "Experiences of Using WDumper to Create Topical Subsets from Wikidata." In: June 2021.

- [29] Jena D. Hwang, Chandra Bhagavatula, Ronan Le Bras, Jeff Da, Keisuke Sakaguchi, Antoine Bosselut, and Yejin Choi. “COMET-ATOMIC 2020: On Symbolic and Neural Commonsense Knowledge Graphs.” In: *AAAI Conference on Artificial Intelligence*. 2020.
- [30] ISO Central Secretary. *Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. en. Standard ISO/IEC TR 9075-1:2016. International Organization for Standardization, 2016. URL: <https://www.iso.org/standard/63555.html>.
- [31] ISO Central Secretary. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*. en. Standard ISO/IEC 9075-2:2016/Cor 2:2022. International Organization for Standardization, 2016. URL: <https://www.iso.org/standard/84487.html>.
- [32] Ali Ismayilov, Dimitris Kontokostas, S. Auer, Jens Lehmann, and Sebastian Hellmann. “Wikidata through the Eyes of DBpedia.” In: *ArXiv abs/1507.04180* (2015).
- [33] Srini Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. “Summarizing Source Code using a Neural Attention Model.” In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016).
- [34] Labra-Gayo Je et al. “Knowledge graphs and wikidata subsetting.” In: 2021.
- [35] N.F. Johnson. *Two’s Company, Three is Complexity: A Simple Guide to the Science of All Sciences*. A Oneworld book. Oneworld, 2007. ISBN: 9781851684885. URL: <https://books.google.de/books?id=NnEcaQAIAAJ>.
- [36] M. Klein and Dieter A. Fensel. “Ontology versioning on the Semantic Web.” In: *International Conference on Semantic Web & Web Services*. 2001.
- [37] Andreas Kokkalis, Panagiotis Vagenas, Alexandros Zervakis, Alkis Simitsis, Georgia Koutrika, and Yannis E. Ioannidis. “Logos: a system for translating queries into narratives.” In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012).
- [38] Georgia Koutrika, Alkis Simitsis, and Yannis E. Ioannidis. “Explaining structured queries in natural language.” In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)* (2010), pp. 333–344.
- [39] Ayush Kumar, Parth Nagarkar, Prabhav Nalhe, and Sanjeev Vijayakumar. “Deep Learning Driven Natural Languages Text to SQL Query Conversion: A Survey.” In: *ArXiv abs/2208.04415* (2022).
- [40] Chris van der Lee, Albert Gatt, Emiel van Miltenburg, and Emiel J. Kraemer. “Human evaluation of automatically generated text: Current trends and best practice guidelines.” In: *Comput. Speech Lang.* 67 (2021), p. 101151.
- [41] Jens Lehmann et al. “DBpedia - A large-scale, multilingual knowledge base extracted from Wikipedia.” In: *Semantic Web 6* (2015), pp. 167–195.

- [42] Douglas B. Lenat. "CYC: a large-scale investment in knowledge infrastructure." In: *Commun. ACM* 38 (1995), pp. 32–38.
- [43] Da Ma, Xingyu Chen, Ruisheng Cao, Zhi Chen, Lu Chen, and Kai Yu. "Relation-Aware Graph Transformer for SQL-to-Text Generation." In: *Applied Sciences* (2021).
- [44] E. Marchi and O. Miguel. "On the structure of the teaching-learning interactive process." In: *International Journal of Game Theory* 3.2 (1974), pp. 83–99. ISSN: 1432-1270. DOI: [10.1007/BF01766394](https://doi.org/10.1007/BF01766394). URL: <https://doi.org/10.1007/BF01766394>.
- [45] Raina Mason, Carolyn Seton, and Graham Cooper. "Applying cognitive load theory to the redesign of a conventional database systems course." In: *Computer Science Education* 26 (Mar. 2016). DOI: [10.1080/08993408.2016.1160597](https://doi.org/10.1080/08993408.2016.1160597).
- [46] T.J. McCabe. "A Complexity Measure." In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. DOI: [10.1109/TSE.1976.233837](https://doi.org/10.1109/TSE.1976.233837).
- [47] Daphne Miedema, Efthimia Aivaloglou, and G. Fletcher. "Identifying SQL Misconceptions of Novices: Findings from a Think-Aloud Study." In: *Proceedings of the 17th ACM Conference on International Computing Education Research* (2021).
- [48] Daphne Miedema, Efthimia Aivaloglou, and G. Fletcher. "Identifying SQL misconceptions of novices." In: *ACM Inroads* 13 (2022), pp. 52–65.
- [49] Daphne Miedema, G. Fletcher, and Efthimia Aivaloglou. "Expert Perspectives on Student Errors in SQL." In: *ACM Transactions on Computing Education (TOCE)* (2022).
- [50] Daphne Miedema, G. Fletcher, and Efthimia Aivaloglou. "So many brackets! An analysis of how SQL learners (mis)manage complexity during query formulation." In: *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)* (2022), pp. 122–132.
- [51] George A. Miller. "WordNet: A Lexical Database for English." In: *Commun. ACM* 38 (1995), pp. 39–41.
- [52] Masahiro Naito, Sho Yokoi, Geewook Kim, and Hidetoshi Shimodaira. "Revisiting Additive Compositionality: AND, OR and NOT Operations with Word Embeddings." In: *ArXiv abs/2105.08585* (2021).
- [53] Heiko Paulheim. "Towards Profiling Knowledge Graphs." In: *PROFILESISWC*. 2017.
- [54] Sini Govinda Pillai, Lay-Ki Soon, and Su-Cheng Haw. "Comparing DBpedia, Wikidata, and YAGO for Web Information Retrieval." In: *Intelligent and Interactive Computing* (2019).
- [55] Roel Poppinga. "Knowledge Graphs and Network Text Analysis." In: *Social Science Information* 42.1 (2003), pp. 91–106. DOI: [10.1177/0539018403042001798](https://doi.org/10.1177/0539018403042001798). eprint: <https://doi.org/10.1177/0539018403042001798>. URL: <https://doi.org/10.1177/0539018403042001798>.

- [56] Seth Poulsen, Liia Butler, Abdussalam Alawini, and Geoffrey L. Herman. "Insights from Student Solutions to SQL Homework Problems." In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (2020).
- [57] Bowen Qin et al. "A Survey on Text-to-SQL Parsing: Concepts, Methods, and Future Directions." In: *ArXiv abs/2208.13629* (2022).
- [58] Ehud Reiter and Robert Dale. "Building applied natural language generation systems." In: *Natural Language Engineering* 3 (1997), pp. 57–87. DOI: [10.1017/S1351324997001502](https://doi.org/10.1017/S1351324997001502).
- [59] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter." In: *ArXiv abs/1910.01108* (2019).
- [60] Yasin N. Silva, Alexis Loza, and Humberto Luiz Razente. "DBSnap-Eval: Identifying Database Query Construction Patterns." In: *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1* (2022).
- [61] Alkis Simitsis and Yannis E. Ioannidis. "DBMSs Should Talk Back Too." In: *ArXiv abs/0909.1786* (2009).
- [62] Amit Singhal. *Introducing the knowledge graph: Things, not strings*. 2012. URL: <https://www.blog.google/products/search/introducing-knowledge-graph-things-not/>.
- [63] John B. Smelcer. "User errors in database query composition." In: *Int. J. Hum. Comput. Stud.* 42 (1995), pp. 353–381.
- [64] Robyn Speer, Joshua Chin, and Catherine Havasi. "ConceptNet 5.5: An Open Multilingual Graph of General Knowledge." In: *ArXiv abs/1612.03975* (2016).
- [65] Emil Stankov, Anastasia Bogdanova, Bojan Ilijoski, and Mile Jovanov. "A SURVEY ON SOFTWARE COMPLEXITY METRICS IN THE CONTEXT OF THEIR APPLICATION IN EDUCATIONAL ENVIRONMENT." In: 2018.
- [66] Made Agus Putra Subali and Siti Rochimah. "A new model for measuring the complexity of SQL commands." In: *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)* (2018), pp. 1–5.
- [67] Toni Taipalus. "Explaining Causes Behind SQL Query Formulation Errors." In: Oct. 2020. DOI: [10.1109/FIE44824.2020.9274114](https://doi.org/10.1109/FIE44824.2020.9274114).
- [68] Toni Taipalus. "The Effects of Database Complexity on SQL Query Formulation (journal-first)." In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2020), pp. 185–185.
- [69] Toni Taipalus and Piia M. H. Perälä. "What to Expect and What to Focus on in SQL Query Teaching." In: *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (2019).

- [70] Thomas Pellissier Tanon, Gerhard Weikum, and Fabian M. Suchanek. "YAGO 4: A Reason-able Knowledge Base." In: *The Semantic Web* 12123 (2020), pp. 583–596.
- [71] Thomas Tanon, Denny Vrandečić, Sebastian Schaffert, Thomas Steiner, and Lydia Pintscher. "From Freebase to Wikidata: The Great Migration." In: Apr. 2016, pp. 1419–1428. DOI: [10.1145/2872427.2874809](https://doi.org/10.1145/2872427.2874809).
- [72] Rahman Taufik and Dade Nurjanah. "An Intelligent Tutoring System with Adaptive Exercises Based on a Student's Knowledge and Misconception." In: *2019 IEEE International Conference on Engineering, Technology and Education (TALE)* (2019), pp. 1–5.
- [73] Aditya Vashistha. "Measuring Query Complexity in SQLShare Workload." In: 2015.
- [74] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention is All you Need." In: *ArXiv abs/1706.03762* (2017).
- [75] Maria-Esther Vidal, Kemele M. Endris, Samaneh Jazashoori, Ahmad Sakor, and Ariam Rivas. "Transforming Heterogeneous Data into Knowledge for Personalized Treatments—A Use Case." In: *Datenbank-Spektrum* 19 (2019), pp. 95–106.
- [76] Lucas Donizetti Vieira. "Accessing related topics through community detection in knowledge graph." In: 2020.
- [77] Denny Vrandečić and Markus Krötzsch. "Wikidata." In: *Communications of the ACM* 57 (2014), pp. 78–85.
- [78] Wei Wei, Bei Zhou, and G. Leontidis. "A Hybrid Natural Language Generation System Integrating Rules and Deep Learning Algorithms." In: *ArXiv abs/2006.09213* (2020).
- [79] Xander Wilcke, Peter Bloem, and Viktor de Boer. "The knowledge graph as the default data model for learning on heterogeneous knowledge." In: *Data Sci.* 1 (2017), pp. 39–57.
- [80] Kun Xu, Lingfei Wu, Zhiguo Wang, Mo Yu, Liwei Chen, and Vadim Sheinin. "SQL-to-Text Generation with Graph-to-Sequence Model." In: *Conference on Empirical Methods in Natural Language Processing*. 2018.
- [81] Sheng Yu and Shijie Zhou. "A survey on metric of software complexity." In: *2010 2nd IEEE International Conference on Information Management and Engineering* (2010), pp. 352–356.
- [82] Kwok bun Yue. "Using a Semi-Realistic Database to Support a Database Course." In: *J. Inf. Syst. Educ.* 24 (2013), pp. 327–336.
- [83] Hongming Zhang, Xin Liu, Haojie Pan, Yangqiu Song, and Cane Wing ki Leung. "ASER: A Large-scale Eventuality Knowledge Graph." In: *Proceedings of The Web Conference 2020* (2019).



- [84] Chris van der Lee, Albert Gatt, Emiel van Miltenburg, and Emiel Kraemer. "Human evaluation of automatically generated text: Current trends and best practice guidelines." In: *Computer Speech & Language* 67 (2021), p. 101151. ISSN: 0885-2308. DOI: <https://doi.org/10.1016/j.csl.2020.101151>. URL: <https://www.sciencedirect.com/science/article/pii/S088523082030084X>.