# S2CLite: a lightweight ontology-agnostic SPARQL to Cypher parser

**Anonymous ACL submission**

## Abstract

We present S2CLite, a lightweight, ontology-agnostic parser that translates SPARQL queries into Cypher queries, enabling both in-situ and large-scale SPARQL to Cypher translation. Unlike existing solutions, S2CLite is purely rule-based (inspired by traditional programming language compilers) and operates without requiring an RDF graph or external tools. Experiments conducted on the BSBM42 and Spider4SPARQL datasets show that S2CLite significantly reduces query parsing errors, achieving a total parsing accuracy of 77.8% on Spider4SPARQL compared to 44.2% by the state-of-the-art S2CTrans. Furthermore, S2CLite achieved a 96.6% execution accuracy on the intersecting subset of queries parsed by both parsers, outperforming S2CTrans by 7.3%. We further use S2CLite to parse Spider4SPARQL queries to Cypher and generate Spider4SSC, a unified Text-to-Query language (SQL, SPARQL, Cypher) dataset with 4525 unique questions and 3 equivalent sets of 2581 matching queries (SQL, SPARQL and Cypher). We open-source S2CLite for further development on GitHub[1] and provide the clean Spider4SSC dataset for download[2].

## 1 Introduction

Knowledge graphs are well-established structured data models where nodes represent unique entities and oriented edges denote their properties and interactions. In the knowledge graph community, the predominant model is Resource Description Framework (RDF) (Cyganiak et al., 2014), as its SPARQL query language follows highly defined W3C standards. However, in the pursuit of efficiency, labelled property graphs have recently gained more traction in industry, with Neo4j and Cypher being the most prominent property graph store and query language respectively (Francis et al., 2018). Unlike RDF graph, in which every resource, relationship or property is represented as a triple, property graph model defines resource nodes and edges as objects with properties, leading to compact and intuitive representation of structured data (Szárnyas et al., 2018) and generally reduced query execution times (Zhao et al., 2023). This pressing need for deeper interconnection and representation between the two storage systems and languages has been addressed by several approaches. Barrasa and Cowley (2021) develop Neosemantics, a universal strategy for converting an RDF graph into an equivalent property graph. Liu et al. (2021) circumvent the languages entirely, by presenting a unified storage scheme, which both Cypher and SPARQL can operate on. On the language side, A Agrawal et al. (2022) and Zhao et al. (2023) leverage Jena ARQ (Wilkinson, 2006) and Neosemantics to develop pipelines for parsing SPARQL queries into equivalent Cypher queries. While robust in nature, their reliance on external tools and insufficient support for common SPARQL query types lead us to present S2CLite, a lightweight and purely algorithmic SPARQL to Cypher parser.

**S2CLite** operates out of the box, requiring neither a specific database file nor a connection to any data store system. In §3 we discuss the significance of S2CLite in the context of existing research. In §4, we describe S2CLite functionality in detail. In §5 we compare our results to Zhao et al.'s (2023) S2CTrans parser (current SotA) and evaluate both parsers with respect to the error rate of unparsed SPARQL queries (§5.2.1) and execution accuracy of parsed queries (§5.2.2) for BSBM (Bizer and Schultz, 2008) and Spider4SPARQL (Kosten et al., 2023) datasets. In §6 we further discuss the experiment results and conclude our findings. In §7, we present limitations of S2CLite and future work.

[1] see 'code/S2CLite_parser/' folder in supplementary files

[2] https://www.dropbox.com/scl/fi/v4wa82hpe2s5p5uh4bgnj/Spider4SSC.zip?rlkey=rb447tkhjq38ugnd0v8rpt2bh&e=3&st=s94sdngb

## 2 Contributions

### 2.1 S2CLite parser

We propose **S2CLite** for rule-based parsing of SPARQL queries to Cypher. Namely, we design the algorithms for Visitor (§4.2) and Interpreter (§4.3) classes presented in §4 below.

### 2.2 Spider4SSC dataset

Using S2CLite, we parse Spider4SPARQL queries into matching Cypher counterparts and compose **Spider4SSC**, an aggregate Text-to-Query language dataset with **4525 unique** (question, sql, sparql, cypher) **entries** where SQL, SPARQL and Cypher are semantically equivalent queries operating over **159 executable** SQLite/RDF graph databases. We collect the SQL queries from Spider (Yu et al., 2018) and SPARQL queries from Spider4SPARQL (Kosten et al., 2023). See Listing 3 for one sample entry from Spider4SSC dev set and refer to §A for dataset creation details and download.

## 3 Related Work

Due to the semantic differences between Cypher and SPARQL, developing a universal standard for their interoperability poses significant challenges (Liu et al., 2021).

### 3.1 SPARQL to Cypher Conversion Tools

This section reviews existing SPARQL to Cypher conversion tools and highlights the key limitations they face.

#### 3.1.1 S2C transpiler

Developed by A Agrawal et al. (2022), SPARQL to Cypher transpiler uses Apache Jena ARQ (Wilkinson, 2006) to generate an Abstract Syntax Tree (AST), which is then traversed by their custom Visitor class[3]. S2C transpiler supports only basic graph patterns, limiting its applicability to queries without any solution modifiers. Due to this limitation, we do not consider it in our experimental section (§5).

#### 3.1.2 S2CTrans

Introduced more recently by Zhao et al. (2023), S2CTrans demonstrates a relational algebra-compliant semantic equivalence between SPARQL and Cypher graph queries. S2CTrans pipeline is robust, leveraging Jena ARQ, Neo4j Neosemantics (Barrasa and Cowley, 2021) and Cypher DSL (Meier and Simons, 2021) in tandem with custom Pattern Matching and Solution Modifier modules to produce syntactically and semantically equivalent Cypher queries. However, this approach presents significant limitations:

1. **Reliance on RDF graph:** S2CTrans requires the RDF graph data file and a connection to Neo4j's Neosemantics. Specifically, S2CTrans loads RDF data via Neosemantics to access the ontology of node properties and relationships. This dependency makes S2CTrans unsuitable for standalone SPARQL to Cypher conversion, as it necessitates the availability of the corresponding RDF knowledge graph.

2. **Limited query types:** S2CTrans supports a substantial amount of SPARQL query types, including complex property paths (`subj p1/p2/p3 obj`) and optional patterns (OPTIONAL), but lacks support for other fundamental syntax, such as solution grouping (GROUP BY) or projection of all bound variables (SELECT *).

**S2CLite** addresses the above limitations with a purely algorithmic solution that operates independently of any ontology or knowledge graph.

### 3.2 Datasets

To asses and compare S2CLite with existing work, we use the two datasets below.

#### 3.2.1 BSBM42

Originally presented by Bizer and Schultz (2008), the Berlin SPARQL Benchmark (BSBM) was designed for evaluating the performance of different RDF graph storage systems on large and complex graph. **BSBM42** contains 42 unique queries (executable on the BSBM knowledge graph) designed by Zhao et al. (2023) to test their S2CTrans parser. In §5.1, we evaluate S2CLite on the same set of queries.

#### 3.2.2 Spider4SPARQL

Spider dataset is a large scale Text-to-SQL composite of 200 SQLite relational databases and 5693 unique SQL queries, paired with 10181 natural language questions (Yu et al., 2018). **Spider4SPARQL** is a sibling dataset aimed at Text-to-SPARQL. Kosten et al. (2023) materialised a large

[3] https://github.com/kracr/sparql-cypher-transpiler

portion of the Spider SQLite relational databases (166 in total) into RDF graphs using the Ontop (Calvanese et al., 2017) Virtual Knowledge Graph system. They further used SemQL, a context-free grammar, as an intermediary language between SQL and SPARQL to facilitate their translation, producing 4721 SPARQL queries, equivalent in functionality to their respective SQL queries. In §5.2, we use Spider4SPARQL to evaluate both S2CTrans and S2CLite with respect to parsing errors and execution accuracy.

## 4 Methods

As previously stated, S2CLite is an algorithmic SPARQL to Cypher parser. Given a SPARQL query, S2CLite generates a semantically equivalent Cypher query. The complete parsing pipeline is depicted in Figure 1. Our approach takes inspiration from traditional programming language compilers, where initially a parse tree is generated from the query to catch any syntax errors, followed by a walker class (visitor or listener), extracting relevant information from the parse tree, which is then interpreted into the output language.

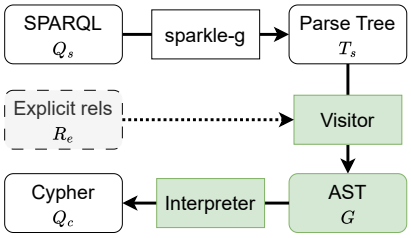


Figure 1: Overview of S2CLite parser. SPARQL query $Q_s$ is transformed into SPARQL 1.1 Parse tree $T_s$ using sparkle-g (2013) grammar. Visitor traverses $T_s$ and generates a Cypher pattern abstract syntax tree ($\mathcal{G}$), which is then used by Interpreter to build the final Cypher query $Q_c$. Visitor can optionally take a set of explicit relationship type names ($R_e$), enforcing them as relationships instead of node properties (see §C).

**0. Parse tree ($T_s$)** of a given SPARQL query is generated using sparkle-g (see §B). This guarantees that the query conforms to SPARQL 1.1 syntax. See Fig. 6 for parse tree example.

**1. Visitor** walks the nodes of $T_s$ and maps visited SPARQL patterns (SELECT, WHERE, etc.) to their respective Cypher equivalents (RETURN, MATCH, resp.), generating an Abstract Syntax Tree (AST, $\mathcal{G}$).

**2. Interpreter** builds the final Cypher query from the patterns and solution modifiers in $\mathcal{G}$.

### 4.1 SPARQL vs. Cypher

Before detailing the Visitor and Interpreter functionality, it is prudent to discuss the general mapping of SPARQL language (RDF graph) to Cypher language (property graph). We will only focus on the necessities for this paper. For in-depth mapping of Cypher using relational algebra, refer to (Marton et al., 2017; Zhao et al., 2023).

Table 1: Overview of mapping SPARQL patterns and solution modifiers to their Cypher equivalent where $e_i$ stands for any expression (variable, literal or function) and $\mathcal{C}$ marks any constraint, such as $a < b$ or $a$ `IN` $b$ etc.

| SPARQL | Cypher |
|---|---|
| SELECT ($e_1$ AS ?x) ?y | RETURN $e_1$ AS x, y |
| WHERE { ?x a :L . ?x :rel ?y .} | MATCH (x:L)-[:rel]->(y) |
| FILTER $\mathcal{C}(e_1, [e_2])$ | WHERE $\mathcal{C}(e_1, [e_2])$ |
| SELECT $e_1$ AS ?x, [$e_2$ AS y]<br>HAVING $\mathcal{C}$(?x, [?y]) | WITH $e_1$ AS x, [$e_2$ AS y]<br>WHERE $\mathcal{C}$(x, [y])<br>RETURN x, [y] |
| SELECT ?x (AVG(?y) AS ?avg)<br>GROUP BY ?x | WITH x, AVG(y) AS avg<br>RETURN x, avg |
| ORDER BY ASC(?x) DESC(?y) | ORDER BY x ASC, y DESC |
| LIMIT n | LIMIT n |
| OFFSET n | SKIP n |

**Mapping SPARQL to Cypher:** In Table 1 we show a general mapping of SPARQL base clauses to their Cypher equivalent. Below are the main syntactic discrepancies between the languages.

**Triple patterns** in WHERE block of SPARQL map to MATCH in Cypher. This is directly related to the basic pattern in the respective graphs. While in RDF graphs, the most atomic pattern is represented by a single triple (subject, predicate, object), in property graphs, every graph node and edge is an object. Objects can have zero or more types (labels) and have zero or more named properties assigned to them. Consider the SPARQL and parsed Cypher queries in Figure 2 as example.

**WITH clause** is a unique Cypher clause, which can be used to project and manipulate results at any point in the Cypher query. WITH has many uses[4], from which we stress the ability to group results and apply aggregate functions to variables. For instance, '`WITH x, AVG(y) AS avg`' in Tab. 2 projects the averages of `y` (as `avg`), grouped by `x`.

[4] https://neo4j.com/docs/cypher-manual/current/clauses/with/

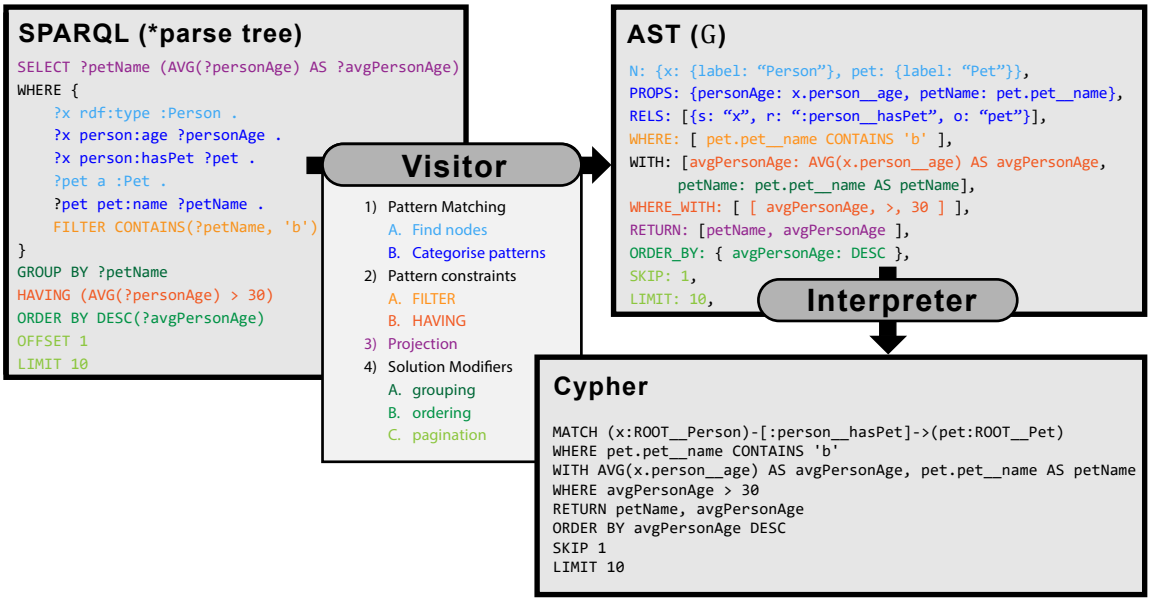


Figure 2: Step by step example of Visitor constructing a simplified version of AST ($\mathcal{G}$) from given SPARQL query and the final Cypher query assembled by the Interpreter. Parse tree creation step is omitted for terseness (see §B). Steps of the Visitor are colour-coded with their respective visited SPARQL pattern and corresponding generated AST entry (see §E for full $\mathcal{G}$). The SPARQL query can be verbalised as: *"Average age of people who own pets with the same name containing the letter 'b', filtered and ordered by owner age and limited to top 10, skipping first one."*

### 4.2 Visitor

Our Visitor is a Python implementation of a Parse tree walker class, sub-classed from ANTLR4 ParseTreeVisitor[5]. Its purpose is to traverse the given parse tree ($T_s$) based on defined logic. At each node of $T_s$, Visitor can add some relevant information to the AST ($\mathcal{G}$, represented as Python dictionary) and decide which child nodes to visit next.

**Building the AST ($\mathcal{G}$):** First, we initialise $\mathcal{G}$ with empty fields using alg. in §D.1. Subsequently, Figure 2 shows the general steps Visitor takes as it builds $\mathcal{G}$ from a given parse tree. We elaborate on the step by step process below.

#### 4.2.1 Pattern Matching

Given a WHERE block with triple patterns $(s\ p\ o) \in \mathcal{T}$, we can divide them into 4 types (see Tab. 2).

Table 2: Types of SPARQL triple patterns and their respective mapping to equivalent Cypher constructs.

| pattern type | triple pattern | cypher pattern |
|---|---|---|
| **1) class label** | ($?x$ `rdf:type` $\mathcal{L}$)<br>($?y$ `a` $\mathcal{L}$) | MATCH ($x$:$\mathcal{L}$)<br>MATCH ($y$:$\mathcal{L}$) |
| **2) value constraint** | ($?x$ `:name 'Emma'`) | MATCH ($x$ {name: 'Emma'}) |
| **3) variable property** | ($?x$ `:age` $?y$) | MATCH ($x$)<br>RETURN $x$.age AS $y$ |
| **4) variable relationship** | ($?x$ `:knows` $?y$) | MATCH ($x$)-[:knows]->($y$) |

[5] https://www.antlr.org/api/Java/org/antlr/v4/runtime/tree/ParseTreeVisitor.html

**Pattern type mapping:** Any variable with a class label (pattern *type 1*) can be mapped as labelled node objects in Cypher ($\mathcal{G}_{\mathcal{N}}[s][L] \leftarrow o$). If object is literal (*type 2*), it can be directly mapped as node property value ($\mathcal{G}_{\mathcal{N}}[s][p] \leftarrow o$). Subsequently, whether given triple should map as a node object property (*type 3*) or a relationship between nodes (*type 4*) requires the prior knowledge of whether $?x$ and $?y$ are nodes in the property graph. Thus, Visitor passes WHERE block two times.

**1.A) Find nodes** detects all *type 1* triple patterns and classifies their subjects as labelled entity nodes (see alg. 1).

**Algorithm 1** First pass (find labelled nodes).

**Require:** $\mathcal{T}$ ▷ all triples in WHERE block
  **for** (s, p, o) $\in \mathcal{T}$ **do**
    **if** p $\in$ {"rdf:type", "a"} **then**
      $\mathcal{G}_{\mathcal{N}}[s][label] \leftarrow o$
    **end if**
  **end for**

**1.B) Categorise patterns** divides all remaining triples ($\mathcal{T}'$) into *type 2*, *type 3* and *type 4*, adding them to $\mathcal{G}_{\mathcal{N}}$, $\mathcal{G}_{\mathcal{P}}$ and $\mathcal{G}_{\mathcal{R}}$ respectively (see alg. 2).

**Algorithm 2** Second pass: Categorise props & rels.

**Require:** $\mathcal{T}'$ (triples), $\mathcal{G}_{\mathcal{N}}$ (nodes), $\mathcal{G}_{\mathcal{V}}$ (all vars)
  $i \leftarrow 0$
  **for** (s, p, o) $\in \mathcal{T}'$ **do**
    **if** $s \in \mathcal{G}_{\mathcal{N}}$ AND $o \in \mathcal{G}_{\mathcal{N}}$ **then**
      $\mathcal{G}_{\mathcal{R}}[i] \leftarrow (s, p, o)$ ▷ add relationship
      $i \leftarrow i + 1$
    **else if** $o \notin \mathcal{G}_{\mathcal{V}}$ **then**
      $\mathcal{G}_{\mathcal{N}}[s][p] \leftarrow o$ ▷ add prop. constraint
    **else**
      $\mathcal{G}_{\mathcal{P}}[o] \leftarrow s.p$ ▷ add var. node property
    **end if**
  **end for**

#### 4.2.2 Constraints

In SPARQL, constraints ($\mathcal{C}$) are applied on matched patterns using either FILTER or HAVING (Lst. 1).

Listing 1: constraint examples

```
ex.1: FILTER ?x IN (10,20,30)
ex.2: HAVING (AVG(?y) < 10)
```

**Categorisation:** While FILTER applies $\mathcal{C}$ before grouping, HAVING applies constraints on grouped results. In contrast, Cypher only has WHERE clause, which can fill the role of both FILTER and HAVING, depending on its position in the query. As discussed in §4.1, using the WITH clause, we can achieve aggregation and grouping equivalent to SPARQL. See alg. 3 for details.

**Algorithm 3** Categorise Constraints

**Require:** $\mathcal{C}$ ▷ constraint in FILTER or HAVING
  $e \leftarrow \mathcal{C}.expr$
  $v \leftarrow \mathcal{C}.var$
  **if** INFILTER($\mathcal{C}$) **then**
    $\mathcal{G}_{Where} \leftarrow \mathcal{C}$
  **else if** INHAVING($\mathcal{C}$) **then**
    **if** $v \notin \mathcal{G}_{WITH}$ **then**
      $\mathcal{G}_{WITH}[v] \leftarrow$ "$e$ AS $v$"
    **end if**
    $\mathcal{G}_{WhereWith} \leftarrow \mathcal{C}$
  **end if**

**2.A) FILTER:** If WHERE is applied before any WITH statement, it is equivalent to using FILTER. When $\mathcal{C}$ is in FILTER clause, the full constraint is passed into $\mathcal{G}_{Where}$ as in eq. (1).

$$G_{Where} \underset{append}{\leftarrow} [x,\ \texttt{IN},\ (10, 20, 30)] \tag{1}$$

**2.B) HAVING:** If WHERE is applied after a WITH statement, it has an equivalent effect to using HAVING. If $\mathcal{C}(e)$ is within a HAVING clause, the expression ($e$) is aliased in $\mathcal{G}_{WITH} \leftarrow e$ `AS a` and the $\mathcal{C}(a)$ is passed to $\mathcal{G}_{WhereWith} \leftarrow \mathcal{C}(a)$ as in eq. (2).

$$\begin{aligned} G_{WITH}[\texttt{agg__0}] &\leftarrow \texttt{AVG}(?y)\ \texttt{AS agg__0} \\ G_{WhereWith} &\underset{append}{\leftarrow} [\texttt{agg__0}, <, 10] \end{aligned} \tag{2}$$

where alias `a=agg__0` is assigned by alg. in §D.10.

#### 4.2.3 Projection

For each expression $e$ in SELECT clause, an equivalent is passed to $\mathcal{G}_{RETURN}$. See §D.6 for details.

#### 4.2.4 Solution Modifiers

**4.A) GROUP BY** does not have a direct equivalent in Cypher. Instead of explicit GROUP BY, Cypher groups implicitly based on grouping keys in the RETURN or WITH statements. The WITH statement can be used for intermediate grouping or aggregation of results, allowing to further filter the grouped results. See §D.7.1 for detailed algorithm.

**4.B) ORDER BY:** Given a set order conditions $\mathcal{O}(e) \in$ `ORDER BY`, where $\mathcal{O}$ can be *ASC*, *DESC* or empty and $e$ is either variable or constraint ($\mathcal{C}$), each is added to $\mathcal{G}_{OB} \leftarrow \{e : \mathcal{O}\}$. If the same constraint was used before and is aliased within $\mathcal{G}_{WITH}$ as $a$, then $\mathcal{G}_{OB} \leftarrow a$. See details in §D.7.2.

**4.C) pagination:** If OFFSET $n$ exists in the SPARQL query, it is added to $\mathcal{G}$ as $\mathcal{G}_{SKIP} \leftarrow n$. If LIMIT $n$ exists in the SPARQL query, it is added to $\mathcal{G}$ as $\mathcal{G}_{LIMIT} \leftarrow n$.

#### 4.2.5 Event handlers

**VisitAggregate:** Whenever an aggregate function, such as (COUNT(?v) AS *?total*) or (AVG(?v)), is encountered, it is added to the $\mathcal{G}_{WITH}$ and $\mathcal{G}_{AGG}$ maps using either explicitly assigned aliases (such as *?total*) or implicitly generated aliases (such as *agg__0*). See alg. D.10.

**VisitVariable:** Whenever a variable ($v$) is encountered by the Visitor, it is cleaned up (stripped of whitespace and any ? or $ characters) and added to the $\mathcal{G}_{\mathcal{V}}$ set of unique variables. If $v$ is part of an aggregate function, the internal variables are automatically replaced by their equivalent namespaced node property.

### 4.3 Interpreter

The Interpreter component is responsible for constructing the final Cypher query from the abstract syntax tree (AST, $\mathcal{G}$) generated by the Visitor (§4.2). This process involves clause-related classes, each designed to build different parts of the Cypher query based on the structured data in $\mathcal{G}$ (see Tab.3).

Table 3: Key classes of the Interpreter, their functions and the parts of $\mathcal{G}$ they use for the clause construction.

| Class | Function | Utilised AST fields |
|---|---|---|
| MATCHBuilder | Constructs MATCH clause (Pattern matching) | $\mathcal{G}_{\mathcal{N}}, \mathcal{G}_{\mathcal{R}}$ |
| RETURNBuilder | Constructs the RETURN clause (Projection) | $\mathcal{G}_{RETURN}$ |
| SMBuilder | Constructs ORDER BY, LIMIT and SKIP | $\mathcal{G}_{OB}, \mathcal{G}_{LIMIT}, \mathcal{G}_{SKIP}$ |
| WHEREBuilder | WHERE clause (non-aggregated constraints) | $\mathcal{G}_{Where}$ |
| WITHBuilder | WITH clause (aggregating and grouping) | $\mathcal{G}_{WITH}$ |
| WWBuilder | second WHERE after WITH (constraints on groups) | $\mathcal{G}_{WhereWith}$ |

#### 4.3.1 Building the Cypher Query

The Interpreter utilises various builder classes to translate $\mathcal{G}$ into a Cypher query. Each builder class is initialised with $\mathcal{G}$ and generates its respective part of the Cypher query. The final Cypher query is assembled by concatenating the outputs of the builders in the correct order (see Listing 2).

Listing 2: Build Cypher query from given AST ($\mathcal{G}$).

```
def parse_ast_to_cypher(g: dict):
    match = MATCHBuilder(g).get()
    where = WHEREBuilder(g).get()
    wth = WITHBuilder(g).get()
    where_wth = WWBuilder(g).get()
    unwind = UNWINDBuilder(g).get()
    ret = RETURNBuilder(g).get()
    sm = SMBuilder(g).get_all()

    return "\n".join(c
        for c in [match, where,
        wth, where_wth, unwind,
        ret, sm] if c).strip("\n")
```

**Interpreter implementation** of the individual Builder classes is detailed in appendix §F.

## 5 Experiments

In the following sections we compare the SPARQL-to-Cypher parsing capabilities of S2CTrans (Zhao et al., 2023) and our S2CLite parser. We focus on two types of experiments:

**Parsing error rate ($\mathcal{E}$):** We run the parsers on all available SPARQL queries of given dataset and evaluate errors, which result in failure to parse the query into Cypher query (i.e. if no Cypher query was produced, it is accounted as parsing error). See Tab. 4 for parsing error groups (based on prominent query types) and Tab. 5 for parsing error results.

**Execution accuracy ($\alpha$):** We collect all SPARQL queries that were successfully parsed into Cypher and evaluate mismatch between the outputs when the pair is executed on their respective knowledge graph (SPARQL on RDFlib[6], Cypher on Neo4j). See Tab. 6 for match error types and Tab. 8 for execution accuracy results.

### 5.1 BSBM42

As S2CTrans and S2CLite were designed with BSBM42 in mind, both parsed all (42) BSBM42 queries into a syntactically valid Cypher commands (see Tab. 5). Furthermore, regarding execution accuracy (Tab. 8), i.e. the percentage of exact match between results of SPARQL query and parsed Cypher query when executed on their respective knowledge graphs, S2CTrans has no execution errors on BSBM42, while S2CLite has one `VAL` error, where row values do not match the row in SPARQL result rows. The respective query, where this error occurs is *OptionalRelationship*. We provide the parsed queries, their outputs and analysis in §G.1.1.

### 5.2 Spider4SPARQL

Spider4SPARQL dataset contains 6144 usable queries with a significantly larger domain of query types compared to BSBM42. As such, we expect both parsing error rate increase and execution accuracy drop for both S2CTrans and S2CLite.

**Symbols:** Consider $N^{Dev}$ (994) and $N^{Train}$ (5150) to be the total amount of queries in the respective Spider4SPARQL (S4S) split. $\mathcal{E}^{Split}_{Tool}$ (e.g. $\mathcal{E}^{Train}_{Lite}$) mark the percentage rate of parsing errors (§5.2.1), while $\alpha^{Split}_{Tool}$ (e.g. $\alpha^{Dev}_{Trans}$) symbolise the execution accuracy (§5.2.2) of the respective S4S splits and parsing tools. Using $\Delta$, change always goes from S2CTrans to S2CLite ($\Delta X = X_{Trans} - X_{Lite}$).

#### 5.2.1 Parsing errors

Refer to Tab. 5 for parsing error rate and the counts of parsing errors for individual query group types. We can observe that S2CLite parses significantly more SPARQL queries without any parsing errors, reducing the error rate by $\Delta\mathcal{E}^{Dev} = 39.5\%$ and $\Delta\mathcal{E}^{Train} = 27.4\%$. S2CTrans is largely limited by

[6] https://rdflib.readthedocs.io/en/stable/

not supporting the `COUNT(*)` projection aggregation. Both S2CTrans and S2CLite show significant parsing error counts when encountering structured queries like nested SELECT clauses (NS2 in Tab. 5) with S2CLite being able to parse about 50% more into Cypher. Finally, both parsers have comparable error counts when parsing queries with constraint expressions such as `NOT EXISTS` and `IN` (NS1 in Tab. 5), indicating the lack of their support.

Table 4: Parsing error types representing query groups.

| error type | query types included |
|---|---|
| COUNT_ALL | queries with COUNT(*) aggregator in SELECT clause |
| NS2 | query structures (nested SELECT, MINUS, COUNT(*) outside of projection) |
| NS1 | queries with constraint expressions (NOT EXISTS, IN/NOT IN, Contains) |
| OTHER | any other unparsed queries |

Table 5: Parsing error types and their counts for BSBM42 and Spider4SPARQL DEV ($N^{Dev} = 994$ queries) and TRAIN sets ($N^{Train} = 5150$ queries). See Tab. 4 for query types included in error categories.

| | BSBM (42) | | S4S DEV (994) | | S4S TRAIN (5150) | |
|---|---|---|---|---|---|---|
| | Trans | Lite | Trans | Lite | Trans | Lite |
| **COUNT_ALL** | 0 | 0 | 278 | **0** | 1042 | **0** |
| **NS2** | 0 | 0 | 175 | **75** | 791 | **404** |
| **NS1** | 0 | 0 | 62 | **47** | 346 | **295** |
| **OTHER** | 0 | 0 | 0 | 0 | **31** | 43 |
| **err rate $\mathcal{E}$ (%)** | 0.0% | 0.0% | 52.0% | 12.5% | 43.4% | 16.0% |

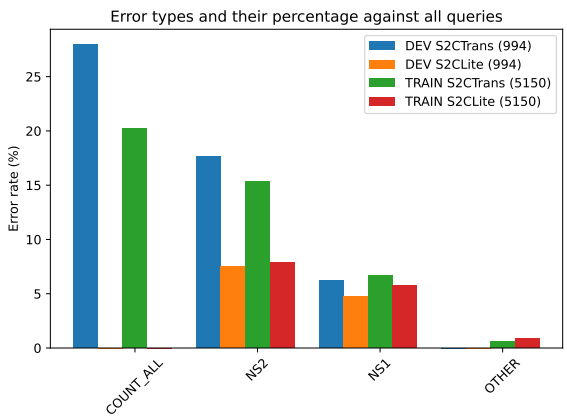


Figure 3: % of parse errors against the total number of SPARQL queries passed to the parser ($N^{Dev} = 994$, $N^{Train} = 5150$). See Tab. 4 for error categories.

### 5.2.2 Execution accuracy

We collect all the successfully parsed Cypher queries (resp. to the parser tool) and execute them over the Neo4j property graph. The resulting outputs are compared to the bindings produced by the original SPARQL query. We consider the SPARQL and Cypher query outputs to match (to be equivalent) if there are no mismatch errors. We define three types of mismatch errors (see Tab. 6).

Table 6: Types of mismatch errors when comparing SPARQL and Cypher query outputs. When at least one mismatch error is raised, the outputs do not match.

| error type | mismatch error type description |
|---|---|
| **NUM_RES** | Number of result rows does not match with SPARQL results |
| **VAL** | Value of one or more rows does not match with SPARQL results |
| **N4j_EXEC** | Neo4j raises an error when trying to execute the query |

**Parsed queries ($C_\forall$):** Following experiments in §5.2.1, we collect all successfully parsed Cypher queries from their respective tools. Out of the 994 and 5150 queries, S2CTrans parsed a total of 471 and 2563 queries into Cypher, while S2CLite parsed 936 and 4749 respectively (see Tab. 7).

**Intersection ($C_\cap$):** We also consider a subset of queries, which were parsed successfully by both S2CTrans and S2CLite. This subset accounts to $C_\cap^{Dev} = 467$ and $C_\cap^{Train} = 2563$. When compared to $C_{\forall,Trans}$, we can see that most queries parsed by S2CTrans are included in the intersection ($\Delta C_{\forall-\cap}^{Dev} = 4$ and $\Delta C_{\forall-\cap}^{Train} = 0$). Thus, all but 4 queries parsed by S2CTrans, were also parsed by S2CLite without any parsing errors.

Table 7: Total number of queries ($N$) in respective splits of Spider4SPARQL, total number of Cypher queries parsed without any parsing errors ($C_\forall$) and intersection of queries, parsed successfully by both parsers ($C_\cap$).

| | S4S DEV | | S4S TRAIN | |
|---|---|---|---|---|
| | Trans | Lite | Trans | Lite |
| $N$ | 994 | | 5150 | |
| $C_\forall$ | 471 | 936 | 2563 | 4749 |
| $C_\cap$ | 467 | | 2563 | |

**Execution accuracy:** For a fair comparison of execution accuracy ($\alpha$) between S2CTrans and S2CLite, we conduct two experiments:

1. $\alpha_\forall$: Execute all available parsed Cypher queries ($C_\forall$), count their match errors and calculate relative $\alpha$ against $C_\forall$ and total accuracy ($\tau$) against $N$ (Tab. 8 and Fig. 4).

2. $\alpha_\cap$: Repeat the above for only intersecting queries ($C_\cap$), parsed successfully by both S2CTrans and S2CLite (Tab. 9 and Fig. 5).

**Discussion:** Given the execution match ($M$) in Tab. 8, S2CLite in its current state successfully transforms 4779 out of the total of 6144 given SPARQL queries ($N^{Dev} + N^{Train}$), amounting to **total parsing accuracy** $\tau_{Lite} = 77.8\%$ (compared to $\tau_{Trans} = 44.2\%$) on Spider4SPARQL dataset.

**Considering** $\alpha_\forall$ in Tab. 8, we observe higher execution accuracy in S2CTrans for both DEV and TRAIN, with significant increase in TRAIN ($\Delta\alpha_\forall^{Train} = 6.1\%$). This suggests robust parsing rules and well defined boundaries of S2CTrans, limiting itself only to queries it was designed for. In parallel, Fig. 4 shows that only a minimal portion of match errors in S2CTrans stem from query execution errors (N4j_EXEC), while ca. 7% of parsed queries, return different number of results (NUM_RES) compared to the SPARQL bindings. Conversely, for S2CLite 8 to 13% (DEV, TRAIN resp.) of its parsed Cypher queries fail to execute properly. This suggests that S2CLite generates false positives (syntactically invalid Cypher queries) in the parsing stage.

**Considering** $\alpha_\cap$ in Tab. 9, S2CTrans shows lower execution accuracy in both dataset splits ($\Delta\alpha_\cap^{Dev} = -6.0\%$ and $\Delta\alpha_\cap^{Train} = -7.3\%$). Subsequently, noting a significant reduction of match error rates for S2CLite in Fig. 5, we conclude that S2CLite produces more semantically accurate queries than S2CTrans, when restricted to $C_\cap$.

Table 8: Execution accuracy ($\alpha_\forall$) of **all** parsed queries. $C_\forall$ marks the count of all parsed Cypher queries. $M$ counts how many $C_\forall$ had their result projections match with the SPARQL query results. **err** rows show counts of execution errors (see Tab. 6). Final rows list execution accuracy ($\alpha_\forall = \frac{M}{C_\forall}$) and total accuracy ($\tau = \frac{M}{N}$).

| | BSBM (42) | | S4S DEV (994) | | S4S TRAIN (5150) | |
|---|---|---|---|---|---|---|
| | **Trans** | **Lite** | **Trans** | **Lite** | **Trans** | **Lite** |
| **parsed** ($C_\forall$) | 42 | 42 | 471 | **936** | 2563 | **4749** |
| **exec match** ($M$) | **42** | 41 | 424 | 826 | 2290 | 3953 |
| **err (NUM_RES)** | 0 | 0 | 35 | **9** | 156 | **9** |
| **err (VAL)** | **0** | 1 | **10** | 33 | **90** | 157 |
| **err (N4j_EXEC)** | 0 | 0 | **2** | 68 | **27** | 630 |
| **exec acc** $\alpha_\forall$ **(%)** | **100%** | 97.6% | **90.0%** | 88.2% | **89.3%** | 83.2% |
| **total acc** $\tau$ **(%)** | **100%** | 97.6% | 42.7% | **83.1%** | 44.5% | **76.8%** |

Table 9: Execution accuracy ($\alpha_\cap$) of **intersecting** parsed queries. $C_\cap$ marks queries successfully parsed by both parsers. $M$ counts how many of $C_\cap$ results match with the SPARQL query results. Further rows count error types (see Tab. 6). Final row shows $\alpha_\cap = \frac{M}{C_\cap}$.

| | S4S DEV (994) | | S4S TRAIN (5150) | |
|---|---|---|---|---|
| | **Trans** | **Lite** | **Trans** | **Lite** |
| **parsed** ($C_\cap$) | 467 | 467 | 2563 | 2563 |
| **exec match** ($M$) | 422 | 450 | 2288 | 2477 |
| **err (NUM_RES)** | 35 | **9** | 156 | **7** |
| **err (VAL)** | 8 | **6** | 92 | **47** |
| **err (N4j_EXEC)** | **2** | **2** | **27** | 32 |
| **exec acc** $\alpha_\cap$ **(%)** | 90.4% | **96.4%** | 89.3% | **96.6%** |

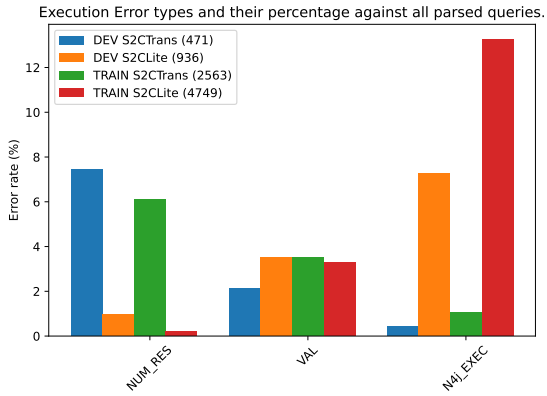


Figure 4: Execution error rate (%) against the total number of **all** successfully parsed Cypher queries ($C_\forall$).

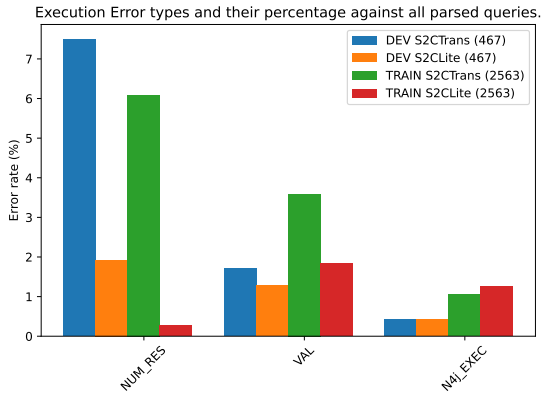


Figure 5: Execution error rate (%) against the total count of **intersecting** parsed Cypher queries ($C_\cap$).

## 6 Conclusion

We presented S2CLite (§4), a stand-alone algorithmic SPARQL to Cypher (S2C) parser, as a lightweight alternative to existing S2C parsers (§3.1) and evaluated its feasibility on BSBM42 (§3.2.1) and Spider4SPARQL (S4S, §3.2.2) datasets. In our experiments (§5), we compare S2CLite to the current SotA parser S2CTrans (§3.1.2). We conclude that S2CLite outperforms S2CTrans both in terms of parsing errors (§5.2.1) and execution accuracy (§5.2.2). In §5.2.1 we show that S2CLite parses a larger subset of query types (ca. 33% error decrease, Tab. 5) but produces significantly more Cypher queries with execution errors (N4j_EXEC in Tab. 8). However, when limited to the intersection of queries between S2CTrans and S2CLite ($C_\cap$, Tab. 7), S2CLite offers 6% execution accuracy increase over S2CTrans (Tab. 9). As a use case, we utilized S2CLite to efficiently parse Spider4SPARQL queries to Cypher and generate the joint Spider4SSC dataset (§A).

## 7 Limitations

**Explicit relationships:** As we discuss in §C, while S2CLite can directly translate any given SPARQL query to Cypher without any third-party dependencies, it is limited in the ability of differentiating between property and relationship patterns between variable nodes. This limitation is alleviated by allowing to explicitly define a set of predicates, which are to be handled as node relationships (shown as $R_e$ in Fig. 1). However, an alternative approach may be desirable, if a file or ontology of the knowledge graph is available at hand.

**Nested queries:** During the parsing error rate evaluation of S2CLite, we show that ca. 8% of the Spider4SPARQL dataset queries are not supported due to the existence of nested queries (see NS2 in Fig. 3). Our aim is to implement support for simple nested queries to eliminate this limitation and further increase the parsing accuracy.

**False positives:** Comparing the error counts between Tab. 8 and Tab. 9 indicates a drastic reduction in S2CLite N4j_EXEC errors, suggesting that many of the false positives (queries that are parsed but not executable) are the ones that are outside of the $C_\forall$ intersection subset. This calls for better definition of explicit constraints and boundaries on the queries that S2CLite supports. In future versions of S2CLite, we aim to include concrete error messages, when unsupported queries are encountered.

**MATCH after WHERE:** As we show in 5.1, S2CLite fails to produce exact match of results for the *OptionalRelationship* SPARQL query (see §G.1.1 for details). This may stem from a known limitation of S2CLite and needs to be addressed further. S2CLite always positions all MATCH patterns at the beginning of the query and does not define logic for positioning MATCH patterns after WHERE, which is required, in this scenario, to produce the exact results match to the SPARQL query execution. mapped

**Limited dataset testing:** One of the foundational aims of S2CLite is the ability to, given a large scale Text-to-SPARQL dataset, produce structurally and semantically identical Text-to-Cypher datasets. By the nature of Text-to-QL, the entries in such datasets need to be covering many domains and query types, which asks for further tests on different Text-to-SPARQL datasets.

**Unique queries:** Our experiments consider all valid entries in the Spider4SPARQL dataset as queries. However, similar to Spider, Spider4SPARQL may contain multiple entries with the same query, paraphrasing the question. Thus, the query counts in our experiments include these duplicate queries.

**Reduced Spider4SSC dataset size:** The generated Spider4SSC dataset is currently limited to 4525 unique entries (4496 unique questions out of the total 7893). This is due to our strict selection process, where we only include (question, sql, sparql, cypher) entries, which return exactly the same results when executed on their respective database. See 10 for our count of questions with exactly matching (*equivalent*) query execution results. The *not equivalent* entries have diverse causes, such as no result from one of the languages, same fields but different values, different number of entries, extra fields bound at the output etc. In the future, we plan to use general LLMs to attempt at fixing mistakes in the queries to further increase the size of the *equivalent* subset and expanding Spider4SSC with newly acquired matching queries.

Table 10: Counts of unique questions with equivalent, not equivalent or empty execution results across all three query languages when executed on their respective database. We use the only the **equivalent** questions to build our Spider4SSC dataset.

| unique questions | dev | train | total |
|---|---|---|---|
| **equivalent** | 608 | 3888 | 4496 |
| **not equivalent** | 402 | 2817 | 3219 |
| **empty** | 22 | 156 | 178 |
| **total** | 1032 | 6861 | 7893 |

## 8 Potential risks

S2CLite is a rule-based parser with limited query support, which was generally aimed to accommodate the available query types in BSBM, Spider and Spider4SPARQL datasets. It is not guaranteed that any arbitrary SPARQL query will be parsed into semantically equivalent Cypher query. With this in mind, if S2CLite is used for automatic dataset creation, further evaluation of the query equivalence such as execution accuracy should be conducted.

## References


Lakshya A Agrawal, Nikunj Singhal, and Raghava Mutharaju. 2022. A sparql to cypher transpiler: Proposal and initial results. In *Proceedings of the 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD)*, pages 312–313.

Jesús Barrasa and Adam Cowley. 2021. neosemantics (n10s): Neo4j rdf semantics toolkit-neo4j labs.

Christian Bizer and Andreas Schultz. 2008. Benchmarking the performance of storage systems that expose sparql endpoints. In *Proc. 4 th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, page 39. Citeseer.

Patrick Blackburn, Claire Gardent, and Wilfried Meyer-Viol. 1993. Talking about trees. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht, The Netherlands. Association for Computational Linguistics.

Diego Calvanese, Benjamin Cogrel, Sarah Komla-Ebri, Roman Kontchakov, Davide Lanti, Martin Rezk, Mariano Rodriguez-Muro, and Guohui Xiao. 2017. Ontop: Answering sparql queries over relational databases. *Semantic Web*, 8(3):471–487.

Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J Carroll, and Brian McBride. 2014. Rdf 1.1 concepts and abstract syntax. *W3C recommendation*, 25(02):1–22.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pages 1433–1445.

Catherine Kosten, Philippe Cudré-Mauroux, and Kurt Stockinger. 2023. Spider4sparql: A complex benchmark for evaluating knowledge graph question answering systems. In *2023 IEEE International Conference on Big Data (BigData)*, pages 5272–5281. IEEE.

Baozhu Liu, Xin Wang, Pengkai Liu, Sizhuo Li, Qiang Fu, and Yunpeng Chai. 2021. Unikg: A unified interoperable knowledge graph database system. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2681–2684.

József Marton, Gábor Szárnyas, and Dániel Varró. 2017. Formalising opencypher graph queries in relational algebra. In *Advances in Databases and Information Systems: 21st European Conference, ADBIS 2017, Nicosia, Cyprus, September 24-27, 2017, Proceedings 21*, pages 182–196. Springer.

Gerrit Meier and Michael Simons. 2021. Neo4j cypher domain-specific language.

Terence J. Parr and Russell W. Quong. 1995. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810.

sparkle-g. 2013. Sparql 1.1 grammar pretty printer and syntactic vallidator.

Gábor Szárnyas, József Marton, János Maginecz, and Dániel Varró. 2018. Reducing property graph queries to relational algebra for incremental view maintenance. *arXiv preprint arXiv:1806.07344*.

K. Wilkinson. 2006. Jena property table implementation. In *Proc. of the 2nd Int'l Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 35–46.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921, Brussels, Belgium. Association for Computational Linguistics.

Zihao Zhao, Xiaodong Ge, Zhihong Shen, Chuan Hu, and Huajin Wang. 2023. S2ctrans: Building a bridge from sparql to cypher. In *International Conference on Database and Expert Systems Applications*, pages 424–430. Springer.

## A Spider4SSC

Spider4SSC (download[7]) is a composite Text-to-Query language dataset with 4525 unique (question, sql, sparql, cypher) entries where semantically matching queries are provided in SQL, SPARQL and Cypher. To build the dataset, we downloaded the Spider4SPARQL dataset dev and train JSONs and materialised RDF knowledge graphs (KGs) following the instructions on Kosten et al.'s (2023) GitHub[8]. Furthermore, we fixed and re-materialised 23 empty KGs and omitted 7 with no SQLite counterpart, resulting in total of 159 valid and equivalent SQLite/RDF graph databases. Using S2CLite (§4), we parsed all available SPARQL queries in Spider4SPARQL into matching Cypher queries. Finally, downloading Spider queries and SQLite databases[9], we assembled the **Spider4SSC** dataset from all executable and valid queries. To ensure the queries are equivalent, all three queries must return matching results when executed over their respective database (using sqlite3[10] for SQL, rdflib 7.0.0[11] for SPARQL and Neo4j with neosemantics[12] for Cypher). We include only question/-query pairs which strictly adhere to this rule. Refer to Listing 3 for an example entry from Spider4SSC dev set. A full technical report detailing the process of creating Spider4SSC will be published at a later time on arXiv.

Listing 3: Sample entry from Spider4SSC dataset.

```
{
"db_id": "concert_singer",
"question": "How many singers do we have?",
"sql": "SELECT count(*) FROM singer",
"sparql": "select (count( *) as ?aggregation_all)
           where { ?t1 a :singer . }",
"cypher": "MATCH (t1:ROOT__singer)
           WITH COUNT(*) AS aggregation_all
           RETURN aggregation_all",
"namespaces": [
    "concert",
    "stadium",
    "singer_in_concert",
    "singer",
    "XMLSchema"
]
},
```

[7] https://www.dropbox.com/scl/fi/v4wa82hpe2s5p5uh4bgnj/Spider4SSC.zip?rlkey=rb447tkhjq38ugnd0v8rpt2bh&e=3&st=s94sdngb

[8] https://github.com/ckosten/Spider4SPARQL

[9] https://yale-lily.github.io/spider

[10] https://docs.python.org/3.11/library/sqlite3.html

[11] https://rdflib.readthedocs.io/en/stable/

[12] (Barrasa and Cowley, 2021)

## B SPARQL parse tree (sparkle-g)

Parse tree ($T_s$) is an ordered tree that formalises the syntactic structure of a given string using specific grammar (Blackburn et al., 1993). For SPARQL string and grammar, each non-terminal (non-leaf) node in $T_s$ represents SPARQL constructs (clause, pattern or function, while terminal (leaf) nodes are reserved for symbols (keywords, literals, variables, operators, etc.). In our work, we use SPARQL 1.1 grammar from the sparkle-g project (sparkle-g, 2013), which adheres to the official W3C SPARQL 1.1 grammar[13]. We use version 4 of the public-domain parser generator ANTLR (Parr and Quong, 1995) to compile a Python-based SPARQL 1.1 grammar parser. We further refer to this compiled parser simply as **sparkle-g**, to honour the original authors. There are several other available options for implementation of SPARQL-based grammar, such as SPARQL 1.0 grammar from the ANTLR4 grammars repository[14] or SPARQL 1.1 grammar from Martin Schröder[15].

### B.1 SPARQL parse tree example

Listing 4 shows the SPARQL query from Fig. 2. The generated parse tree using ANTLR4 parser with sparkle-g grammar is shown in Fig. 6.

Listing 4: SPARQL query example (same as in Fig. 2).

```
SELECT ?petName (AVG(?personAge)
AS ?avgPersonAge)
WHERE {
  ?x rdf:type :Person .
  ?x person:age ?personAge .
  ?x person:hasPet ?pet .
  ?pet a :Pet .
  ?pet pet:name ?petName .
  FILTER CONTAINS(?petName, 'b')
}
GROUP BY ?petName
HAVING (AVG(?personAge) > 30)
ORDER BY DESC(?avgPersonAge)
OFFSET 1
LIMIT 10
```

[13] https://www.w3.org/TR/sparql11-query/#grammar

[14] https://github.com/antlr/grammars-v4/tree/master

[15] https://github.com/schrer/sparql-g4

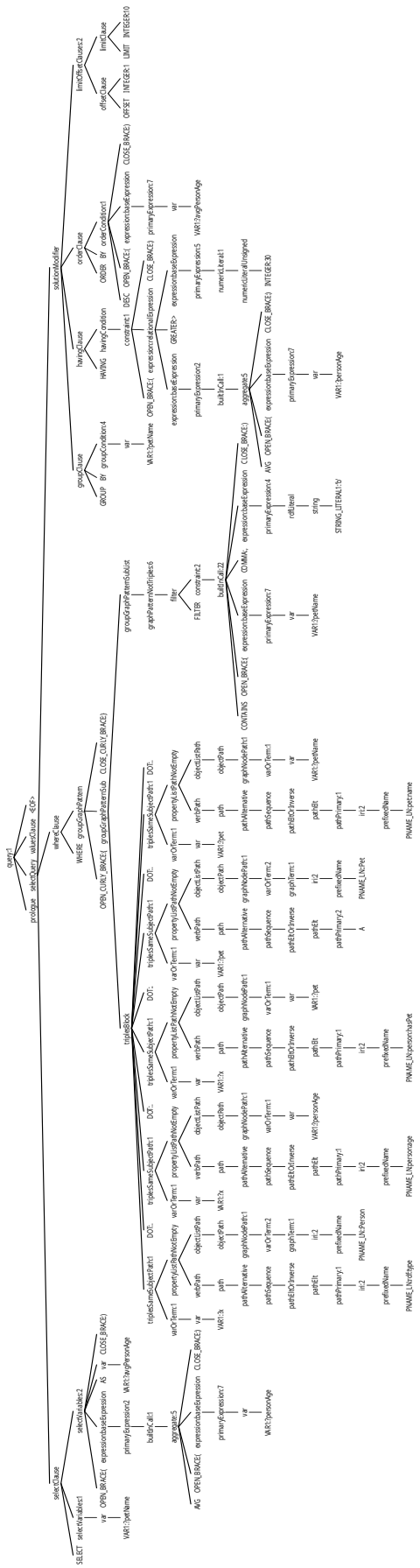

Figure 6: A parse tree of SPARQL clause in Fig. 2 (and Lst. 4), generated using sparkle-g grammar (§B).

## C Explicit relationship type names

In 1 we mention that Visitor can receive optional list of explicit relationships ($R_e$). To understand the necessity for this, consider the example query in Listing 5.

Listing 5: *Average age of all people who Emma knows.*

```
PREFIX : <http://example.org/>
SELECT AVG(?ag) AS ?avgAge
WHERE {
  ?x a :Person .
  ?x :name 'Emma' .
  ?x :knows ?y .
  ?y :age ?ag .
}
```

Note the omission of class definition (type 1 pattern) for ?y (i.e. `(?y a :Person)`). This is a valid SPARQL in which `(?x :knows ?y)` represents a type 4 pattern (see Tab. 2). However, as we discuss in 4.2.1, alg. 2 identifies pattern as type 4 only when `?x` and `?y` are both known node object variables. As alg. 1 will not mark `?y` as node object, the pattern `(?x :knows ?y)` will get categorised as type 3 pattern instead.

**Explicit relationship types** circumvent the default pattern categorisation. Any predicate specified in $R_e$ will automatically set pattern as type 4. For example, if $R_e$ = `(':knows',)`, then `(?x :knows ?y)` is explicitly type 4 and is added to $\mathcal{G}_{\mathcal{R}}$ instead of $\mathcal{G}_{\mathcal{P}}$.

## D Visitor algorithm details

Bellow are pseudocode algorithms of the parsing strategy implemented in our custom Visitor class, which traverses the SPARQL parse tree and builds an abstract syntax tree (AST) represented as a graph $\mathcal{G}$ (Python dictionary class in practice). The graph $\mathcal{G}$ is then used to assemble an equivalent Cypher query. Algorithms used in the main sections are also included again for completeness.

### D.1 Initialise AST Containers

The graph containers are initialised to store various elements encountered during the parsing.

**Algorithm 4** Initialise graph containers

**Ensure:** $\mathcal{G} \leftarrow \{\}$ ▷ empty graph
$\mathcal{G}_{\mathcal{V}} \leftarrow [\,]$ ▷ empty variable list
$\mathcal{G}_{\mathcal{N}} \leftarrow \{\}$ ▷ empty node map
$\mathcal{G}_{\mathcal{P}} \leftarrow \{\}$ ▷ empty node property map
$\mathcal{G}_{\mathcal{R}} \leftarrow [\,]$ ▷ empty relationship list
$\mathcal{G}_{T(\mathcal{R})} \leftarrow [\,]$ ▷ empty relationship types list
$\mathcal{G}_{AGG} \leftarrow \{\}$ ▷ empty map of aggregation functions
$\mathcal{G}_{SUBG} \leftarrow \{\}$ ▷ empty map of subgraphs (not implemented)
$\mathcal{G}_{Where} \leftarrow [\,]$ ▷ empty list of simple constraints
$\mathcal{G}_{With} \leftarrow \{\}$ ▷ empty map of aliases
$\mathcal{G}_{WhereWith} \leftarrow [\,]$ ▷ empty list of aggregated/aliased constraints
$\mathcal{G}_{UNWIND} \leftarrow \{\}$ ▷ empty map of collections to unwind
$\mathcal{G}_{RETURN} \leftarrow [\,]$ ▷ empty list of variables to return
$\mathcal{G}_{OB} \leftarrow \{\}$ ▷ empty map of variables to order by
$\mathcal{G}_{LIMIT} \leftarrow null$ ▷ int container for limiting returned rows
$\mathcal{G}_{SKIP} \leftarrow null$ ▷ int container for skipping returned rows

### D.2 Visit WHERE Clause

The WHERE clause is visited in two passes. The first pass identifies which resources are nodes and labels them. The second pass categorises triples into node properties and relationships.

### D.3 First Pass: Identify Entities and Labels

**Algorithm 5** WHERE Clause Pass 1: Identify Labelled Nodes

**Require:** $\mathcal{T}$ ▷ all triples in query
  **for** (s, p, o) $\in \mathcal{T}$ **do**
    **if** p $\in$ {"rdf:type", "a"} **then**
      $\mathcal{G}_{\mathcal{N}}[s][label] \leftarrow o$
    **end if**
  **end for**

### D.4 Second Pass: Categorise Properties and Relationships

**Algorithm 6** WHERE Clause Pass 2: Categorise Properties and Relationships

**Require:** $\mathcal{T}'$ ... triples, $\mathcal{G}_{\mathcal{N}}$ ... nodes, $\mathcal{G}_{\mathcal{V}}$ ... all variables
  $i \leftarrow 0$
  **for** (s, p, o) $\in \mathcal{T}'$ **do**
    **if** $s \in \mathcal{G}_{\mathcal{N}}$ AND $o \in \mathcal{G}_{\mathcal{N}}$ **then**
      $\mathcal{G}_{\mathcal{R}}[i] \leftarrow (s, p, o)$ ▷ add relationship
      $i \leftarrow i + 1$
    **else if** $o \notin \mathcal{G}_{\mathcal{V}}$ **then**
      $\mathcal{G}_{\mathcal{N}}[s][p] \leftarrow o$ ▷ add node property constraint
    **else**
      $\mathcal{G}_{\mathcal{P}}[o] \leftarrow s.p$ ▷ add variable node property
    **end if**
  **end for**

### D.5 Categorise Constraints

Constraints are divided into two groups. First group accounts for FILTER clauses with no aggregation needed. Second group contains those that operate on projected solutions, amounting to any HAVING clauses or FILTER clauses containing aggregation functions.

**Algorithm 7** visitConstraint

**Require:** $\mathcal{C}$ ▷ constraint in FILTER or HAVING
  $e \leftarrow \mathcal{C}.expr$
  $v \leftarrow \mathcal{C}.var$
  **if** INFILTER($\mathcal{C}$) **then**
    $\mathcal{G}_{Where} \leftarrow \mathcal{C}$
  **else if** INHAVING($\mathcal{C}$) **then**
    **if** $v \notin \mathcal{G}_{WITH}$ **then**
      $\mathcal{G}_{WITH}[v] \leftarrow$ "$e$ AS $v$"
    **end if**
    $\mathcal{G}_{WhereWith} \leftarrow \mathcal{C}$
  **end if**

### D.6 Visit SELECT

The SELECT clause contains all the variables and aliased aggregations required as the output of the query. As Visitor walks every child node, any handling functions such as visitVar (§D.9) or visitAggregate (§D.10) are also triggered.

**Algorithm 8** visitSelectClause

**Require:** $C$ ▷ context of the SELECT clause
  $V \leftarrow [\,]$ ▷ list of projection variables
  $K \leftarrow [\,]$ ▷ list of keywords
  $d \leftarrow False$ ▷ DISTINCT flag
  **for** $ch \in$ GETCHILDREN($C$) **do**
    **if** ISTERMINAL($ch$) **then**
      $t \leftarrow$ UPPER($ch.text$)
      **if** $t$ == "DISTINCT" **then**
        $d \leftarrow True$
      **else if** $t$ == "*" **then**
        $V \leftarrow$ "*"
      **else**
        $K \leftarrow t$
      **end if**
    **else**
      $t_{ch} \leftarrow$ VISIT($ch$)
      **if** $d == True$ **then**
        $V \leftarrow$"DISTINCT $t_{ch}$"
        $d \leftarrow False$
      **else**
        $V \leftarrow t_{ch}$
      **end if**
    **end if**
  **end for**
  $V_{str} \leftarrow$ ", ".join($V$)
  $\mathcal{G}_{RETURN}$.extend($K$)
  $\mathcal{G}_{RETURN}$.append($V_{str}$)

## D.7 Solution Modifiers

The Solution modifier methods further process the output variables, handling grouping (§D.7.1), ordering (§D.7.2) and pagination of result rows with LIMIT (§D.7.3) and OFFSET (§D.7.4).

### D.7.1 Visit GROUP BY

Whenever GROUP BY is encountered, there can be one or more order conditions $c_{GB} \in C_{GB}$. Each $c_{GB}$ can be one of

1. $c_{GB} = e$ `[AS` $?v$`]` (expression & opt. alias)
2. $c_{GB} = f(e)$ (built in or custom function call)
3. $c_{GB} = ?v$ (variable)

Alg. 9 ensures that for each $c_{GB} \in C_{GB}$, group variables are added to the WITH clause and any other RETURN variables, that are not present in $\mathcal{G}_{WITH}$ are called by their name-spaced node property (e.g. $node.name$ instead of $?nodeName$).

**Algorithm 9** visitGroupCondition

**Require:** $c_{GB}$ ▷ one condition in GROUP BY
  $g \leftarrow [\,]$ ▷ group list
  $f_{\mathcal{A}} \leftarrow$ False ▷ flag for alias presence
  **for** $ch \in$ GETCHILDREN$(c_{GB})$ **do**
    **if** ISTERMINAL$(ch)$ **then**
      $t \leftarrow$ UPPER$(ch.text)$
    **else**
      $t \leftarrow$ VISIT$(ch)$
    **end if**
    **if** $t \in$["(", ")"] **then**
      continue ▷ ignore any brackets
    **else if** $t$ == "$AS$" **then**
      $f_{\mathcal{A}} \leftarrow$ True
    **end if**
    $g \underset{app}{\leftarrow} t$ ▷ append to $g$
  **end for**
  **if** $f_{\mathcal{A}}$ == True **then**
    $\mathcal{A} \leftarrow$ POP$(g, -1)$ ▷ pop last element as alias
    $v \leftarrow g[1]$
    **if** $v \notin$ KEYS$(\mathcal{G}_{WITH})$ **then**
      $\mathcal{A}_{dict} \leftarrow \{\mathcal{A} : v$ AS $\mathcal{A}\}$
      UPDATE$(\mathcal{G}_{WITH}, \mathcal{A}_{dict})$
      $g[1] \leftarrow \mathcal{A}$
    **end if**
    POP(g, -1) ▷ remove AS
    $f_{\mathcal{A}} \leftarrow$ False
  **end if**
  $g_{str} \leftarrow$ JOIN(" ", $g$)
  **if** $v \in$ KEYS$(\mathcal{G}_{PROPS})$ **then**
    $\mathcal{G}_{WITH}[v] \leftarrow \mathcal{G}_{PROPS}[v]$ AS $v$
  **end if**
  **return** $g_{str}$

### D.7.2 Visit ORDER BY

When ORDER BY is present in the query, there can be one or more order conditions $c \in C_{OB}$. Each $c$ can be one of

1. $c =$ `(ASC|DESC)`$(e)$ (order and expression)
2. $c = \mathcal{C}$ (constraint)
3. $c = ?v$ (variable)

Alg. 10 applies for each order condition $c_{OB} \in C_{OB}$ and extends the $\mathcal{G}_{OB}$ map, using either existing alias from $\mathcal{G}_{WITH}[c_{OB}]$ or simply passing $c_{OB}$ as is.

**Algorithm 10** visitOrderCondition

**Require:** $c_{OB}$ ▷ one condition in ORDER BY
  $g \leftarrow [\,]$ ▷ order list
  $\mathcal{O} \leftarrow \{\,\}$ ▷ order dictionary
  **if** GETCHILDCOUNT($c_{OB}$) == 1 **then**
    $ch \leftarrow$ VISIT($c_{OB}.child(0)$)
    $\mathcal{O}[ch] \leftarrow$ "ASC" ▷ default sorting
  **else**
    **for** $ch \in$ GETCHILDREN($c_{OB}$) **do**
      **if** ISTERMINAL($ch$) **then**
        $t \leftarrow$ UPPER($ch.text$)
      **else** ▷ expression or variable
        $t \leftarrow$ VISIT($ch$)
      **end if**
      **if** $t$ in ["(", ")"] **then**
        continue ▷ ignore any brackets
      **end if**
      $g \underset{app}{\leftarrow} t$ ▷ append to $g$
    **end for**
  **end if**
  **if** LEN($g$) == 2 **then**
    $\mathcal{O}[g[1]] \leftarrow g[0]$
  **end if**
  $\mathcal{O}_{NS} \leftarrow \{\,\}$ ▷ $\mathcal{O}$ with namespaced node properties
  **for** $k \in$ KEYS($\mathcal{O}$) **do**
    $k_{NS} \leftarrow k$
    **if** $\neg$KEYS($\mathcal{G}_{WITH}$) **and** $k \notin \mathcal{G}_{RETURN}$ **then**
      $k_{NS} \leftarrow \mathcal{G}_{PROPS}[k] \leftarrow k$
    **else if** $k \notin \mathcal{G}_{WITH}$ **then**
      $\mathcal{G}_{WITH}[k] \leftarrow$ "$\mathcal{G}_{PROPS}[k]$ `AS` $k$"
    **end if**
    $\mathcal{O}_{NS}[k_{NS}] \leftarrow \mathcal{O}[k]$
  **end for**
  $\mathcal{G}_{OB} \underset{upd}{\leftarrow} \mathcal{O}_{NS}$ ▷ update ORDER BY map

### D.7.3 Visit LIMIT

**Algorithm 11** visitLimitClause

**Require:** $val$ ▷ LIMIT clause value
  $\mathcal{G}_{LIMIT} \leftarrow$ INT($val$)

### D.7.4 Visit OFFSET

**Algorithm 12** visitOffsetClause

**Require:** $val$ ▷ OFFSET clause value
  $\mathcal{G}_{SKIP} \leftarrow$ INT($val$)

## D.8 Handling Functions

## D.9 Visit Variable

**Algorithm 13** visitVar

**Require:** $v$, $ctx$ ▷ variable name and context
  $v \leftarrow$ STRIP($v$, "?$ ")
  **if** ISINSTANCE($ctx$, `aggregateContext`) **then**
    **if** $v \in \mathcal{G}_{PROPS}$ **then**
      $v \leftarrow \mathcal{G}_{PROPS}[v]$
    **end if**
  **end if**
  $\mathcal{G}_{\mathcal{V}}[i] \leftarrow v$
  return " " + $v$

## D.10 Visit Aggregate or BuiltIn function

**Algorithm 14** visitAggregate

**Require:** $\mathcal{E}$ ... expression, $\mathcal{A}$ ... alias (if exists)
  $i \leftarrow$ LENGTH($\mathcal{G}_{AGG}$)
  **if** NOT EXISTS($\mathcal{A}$) **then**
    $\mathcal{A} \leftarrow$ "agg__{i}"
  **end if**
  $\mathcal{G}_{WITH}[\mathcal{A}] \leftarrow \mathcal{E}$ AS $\mathcal{A}$
  $\mathcal{G}_{AGG}[\mathcal{A}] \leftarrow \mathcal{E}$
  return $\mathcal{A}$

## E AST example

Given the parse tree in Figure 6 (from SPARQL query shown in Figure 2, resp. Listing 4), Visitor produces an abstract syntax tree ($\mathcal{G}$) shown in Listing 6.

Listing 6: AST produced by Visitor by traversing parse tree in Figure 6.

```
{
  "vars": [
    "pet",
    "avgPersonAge",
    "x",
    "petName",
    "personAge"
  ],
  "iri": {},
  "nodes": {
    "x": {
      "label": "ROOT__Person"
    },
    "pet": {
      "label": "ROOT__Pet"
    }
  },
  "props": {
    "personAge": "x.person__age",
    "petName": "pet.pet__name"
  },
  "rels": [
    {
      "s": "x",
      "r": ":person__hasPet",
      "o": "pet",
      "optional": false,
      "inverse": false
    }
  ],
  "rel_types": [
    ":person__hasPet"
  ],
  "aggregates": {
    "avgPersonAge": "AVG(x.person__age)"
  },
  "WHERE": [
    "pet.pet__name CONTAINS 'b'"
  ],
  "WITH": {
    "avgPersonAge": "AVG(x.person__age) AS avgPersonAge",
    "petName": "pet.pet__name AS petName"
  },
  "WHERE_WITH": [
    [
      "avgPersonAge",
      ">",
      "30"
    ]
  ],
  "RETURN": [
    "petName, avgPersonAge"
  ],
  "ORDER BY": {
    "avgPersonAge": "DESC"
  },
  "LIMIT": 10,
  "OFFSET": 1
}
```

## F Interpreter implementation details

Given an arbitrary AST ($\mathcal{G}$), Interpreter defines the builder classes below to construct respective Cypher clauses and concatenates them to produce the final Cypher query. For an example of $\mathcal{G}$, refer to Lst. 6.

### F.1 MATCHBuilder

The MATCHBuilder class constructs the MATCH clause by iterating over the nodes ($\mathcal{G}_{\mathcal{N}}$) and rels ($\mathcal{G}_{\mathcal{R}}$) in the AST to generate Cypher MATCH statements.

Listing 7: MATCHBuilder Class implementation

```
class MATCHBuilder:
    def __init__(self, graph: dict):
        self.g = graph
        self.match = ""
        self.build()

    def build(self):
        nodes = self.g["nodes"]
        for entry in self.g["rels"]:
            s, rel_seq, o = entry["s"], entry["r"], entry["o"]
            s_node = f"({s}:{nodes[s]['label']})"
            o_node = f"({o}:{nodes[o]['label']})"
            rel_str = "-[]-".join([f"[{r}]" for r in rel_seq.split("/")])
            self.match += f"MATCH {s_node}-{rel_str}->{o_node}\n"

    def get(self):
        return self.match.strip()
```

### F.2 RETURNBuilder

The RETURNBuilder class constructs the RETURN clause by utilising the $\mathcal{G}_{RETURN}$ field in the AST to determine which variables and expressions should be projected to the output.

Listing 8: RETURNBuilder Class implementation

```
class RETURNBuilder:
    def __init__(self, graph: dict):
        self.g = graph
        self.return_str = ""
        self.build()

    def build(self):
        term = self.g["RETURN"][0]
        terms = term.split(", ")
        self.return_str = "RETURN " + ", ".join(terms)

    def get(self):
        return self.return_str.strip()
```

### F.3 WHEREBuilder

The WHEREBuilder class constructs the WHERE clause by converting conditions in the $\mathcal{G}_{Where}$ field of the AST into a Cypher-compatible format.

Listing 9: WHEREBuilder Class implementation

```
class WHEREBuilder:
    def __init__(self, graph: dict):
        self.g = graph
        self.term_list = []
        self.where = ""

        for term in self.g["WHERE"]:
            # print(term)
            self.add(term)

        self.build()

    def add(self, term):
        if isinstance(term, dict):
            print(NotImplementedError("Nested SELECT statements not supported yet."))
            return False

        if isinstance(term, list):
            # handle RelationalSetExpressions
            t = _construct_relational_set_expression(term)
        else:
            if "CONTAINS" in term:
                self.term_list.append(term)
                return True

            or_term = term.replace(",", "").replace("( ", "").replace(" )",
                                                                  "").split("||")
            and_or_terms = [orterm.strip().split("&&") for orterm in or_term]
            t = ""
            for subterm in and_or_terms:
                t += "(" + " AND ".join(subterm) + ")" + " OR "

        t = t.removesuffix(" OR ")
        self.term_list.append(t)
        return True

    def build(self):
        where_expr = []
        for term in self.term_list:
            for alias, agg in self.g["aggregates"].items():
                term = term.replace(alias, agg)
            where_expr.append(term)

        if where_expr:
            self.where = "WHERE " + " AND ".join(where_expr)

    def get(self):
        return self.where

    def clear(self):
        self.having_list = []
        self.wth, self.where = (None, None)
```

### F.4 WITHBuilder

WITHBuilder gathers variables and expressions from the $\mathcal{G}_{WITH}$ field in the AST to facilitate aggregation, group variables and aliasing.

Listing 10: WITHBuilder Class implementation

```
class WITHBuilder:
    def __init__(self, graph: dict):
        self.g = graph
        self.with_list = []
        self.wth = ""
        self.build()

    def build(self):
        wth_terms = ", ".join(self.g["WITH"].values())
        self.wth = f"WITH {wth_terms}"

    def get(self):
        return self.wth.strip()
```

### F.5 WWBuilder

The WHERE clause after the first WITH is built using the $\mathcal{G}_{WhereWith}$ part of the AST in Lst. 11. This serves in similar light as HAVING clause in SPARQL (see 4.2.2).

Listing 11: WhereWithBuilder Class implementation

```
class WWBuilder:
    def __init__(self, graph: dict):
        self.g = graph
        self.having_list = []
        self.where = ""

        for term in self.g["WHERE_WITH"]:
            self.add(term)

        self.build()

    def add(self, term):
    term = list(term)
        if isinstance(term, list):
            # handle RelationalSetExpressions
            t = _construct_relational_set_expression(term)
        else:
            or_term = term.replace(",", "").replace("( ", "").replace(" )", "").split("||")
            and_or_terms = [orterm.strip().split("&&") for orterm in or_term]
            t = ""
            for subterm in and_or_terms:
                t += "(" + " AND ".join(subterm) + ")" + " OR "

        t = t.removesuffix(" OR ")
        self.having_list.append(t)

    def build(self):
```

```
        where_expr = []
        if not self.having_list:
            logger.info("No HAVING clause found in the query.")
            return

        for term in self.having_list:
            where_expr.append(term)

        if where_expr:
            self.where = "WHERE " + " AND ".join(where_expr)

    def get(self):
        return f"{self.where}"
```

### F.6 SMBuilder

The Solution Modifier builder class constructs the ORDER BY, LIMIT and SKIP cypher clauses, modifying the solution set of the query.

Listing 12: SMBuilder Class implementation

```
class SMBuilder:
    def __init__(self, graph: dict):
        self.g = graph
        self.order_by = ""
        self.limit = ""
        self.skip = ""
        self.build()

    def build(self):
        if self.g["LIMIT"]:
            self.limit = f"LIMIT {self.g['LIMIT']}"
        if self.g["SKIP"]:
            self.skip = f"SKIP {self.g['SKIP']}"
        if self.g["ORDER BY"]:
            self.order_by = "ORDER BY " + ", ".join(
                [f"{term} {direction}" for term, direction
                 in self.g["ORDER BY"].items()]
                )

    def get_all(self):
        return "\n".join([self.order_by, self.limit, self.skip]).strip()
```

### F.7 Assembling the Final Cypher Query

The final Cypher query is assembled by concatenating the outputs of the builder classes in the correct order. Each builder contributes its respective part to the complete query as shown in Lst. 2 and eq. (3).

$$\text{Cypher} = \text{MATCH} + \text{WHERE} + \text{WITH} + \text{WW} + \text{UNWIND} + \text{RETURN} + \text{SM} \tag{3}$$

# G Error analysis

## G.1 BSBM42

### G.1.1 OptionalRelationship

The *OptionalRelationship* SPARQL query (see 13) was created by Zhao et al. (2023) to test situations, where OPTIONAL block contains relationships (type 4 pattern in 2). In this case, the Cypher query must contain OPTIONAL MATCH clause. Both S2CTrans (see Lst. 14) and S2CLite (see Lst. 15) parse the SPARQL query without major errors. The queries are almost identical, disregarding projection aliasing (which only S2CLite does) and unnecessary restatement of node labels. However, S2CLite fails at this query, as it produces wrong output row values when executed on BSBM. See execution output for the S2CTrans query in Lst. 17 and for S2CLite in Lst. 18 and note that S2CLite does not match to the execution output of original SPARQL query in Lst. 16.

**Cause** of the discrepancy in outputs, is visible from the two parsed Cypher queries. S2CLite puts OPTIONAL MATCH in front of WHERE, while S2CTrans puts if after the WHERE statement. The original SPARQL query has OPTIONAL block before the FILTER, which would syntactically favour the S2CLite version. Contrary to our intuition, S2CTrans produces the correct results while swapping the original order of the equivalent clauses (first WHERE then OPTIONAL MATCH vs first OPTIONAL block then FILTER).

Listing 13: *SPARQL query "OptionalRelationship.rq" from BSBM42 crafted by Zhao et al. (2023)*

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ele: <http://purl.org/dc/elements/1.1/>

SELECT distinct ?label WHERE{
        ?review rdf:type bsbm:Review .
        ?review bsbm:reviewFor ?product .
        ?product rdf:type bsbm-inst:ProductType1.
        ?product bsbm:productPropertyNumeric1 ?pPN1 .
        ?product rdfs:label ?label.
        OPTIONAL {
                ?product ele:publisher ?producer .
                ?producer a bsbm:Producer .
        }
        FILTER(?pPN1 > 1000)
}
ORDER BY(?label)
LIMIT 10
```

Listing 14: *S2CTrans Cypher query parsed from SPARQL Lst. 13 (OptionalRelationship).*

```
MATCH (review:bsbm__Review)-[reviewFor:bsbm__reviewFor]->
(product:bsbm_inst__ProductType1)
WHERE product.bsbm__productPropertyNumeric1 > 1000
OPTIONAL MATCH (product)-[publisher:ele__publisher]->
(producer:bsbm__Producer)
RETURN DISTINCT product.rdfs__label
ORDER BY product.rdfs__label ASC
LIMIT 10
```

Listing 15: *S2CLite Cypher query parsed from SPARQL Lst. 13 (OptionalRelationship).*

```
MATCH (review:bsbm__Review)-[:bsbm__reviewFor]->
(product:bsbm_inst__ProductType1)
OPTIONAL MATCH (product:bsbm_inst__ProductType1)-[:ele__publisher]->
(producer:bsbm__Producer)
WHERE product.bsbm__productPropertyNumeric1 > 1000
RETURN DISTINCT product.rdfs__label AS label
ORDER BY product.rdfs__label ASC
LIMIT 10
```

Listing 16: *Output rows of original SPARQL query in Lst. 13 (OptionalRelationship).*

```
[{'label': 'abjured'},
{'label': 'absorbs'},
{'label': 'abysm'},
{'label': 'abysms'},
{'label': 'accommodator'},
{'label': 'acidness'},
{'label': 'acquirement'},
{'label': 'adorned'},
{'label': 'adsorptiveness'},
{'label': 'adulterously'}]
```

Listing 17: *Output rows of S2CTrans Cypher query parsed from SPARQL Lst. 13 (OptionalRelationship).*

```
[{'label': 'abjured'},
{'label': 'absorbs'},
{'label': 'abysm'},
{'label': 'abysms'},
{'label': 'accommodator'},
{'label': 'acidness'},
{'label': 'acquirement'},
{'label': 'adorned'},
{'label': 'adsorptiveness'},
{'label': 'adulterously'}]
```

Listing 18: *Output rows of S2CLite Cypher query parsed from SPARQL Lst. 13 (OptionalRelationship).*

```
[{'label': 'aardvark'},
{'label': 'ab areaways'},
{'label': 'abdicable'},
{'label': 'abjured'},
{'label': 'ablatively'},
{'label': 'abnegators'},
{'label': 'abnormality'},
{'label': 'abolishment'},
{'label': 'abolitionism'},
{'label': 'abominated'}]
```