# ATHENA++: Natural Language Querying for Complex Nested SQL Queries

Jaydeep Sen[1], Chuan Lei[2], Abdul Quamar[2], Fatma Özcan[2], Vasilis Efthymiou[2],
Ayushi Dalmia[1], Greg Stager[3], Ashish Mittal[1], Diptikalyan Saha[1],
Karthik Sankaranarayanan[1]
[1]IBM Research - India, [2]IBM Research - Almaden, [3]IBM Canada
[1]jaydesen|adalmi08|arakeshk|diptsaha|kartsank@in.ibm.com,
[2]fozcan|ahquamar@us.ibm.com, [2]chuan.lei|vasilis.efthymiou@ibm.com,
[3]gstager@ca.ibm.com

## ABSTRACT

Natural Language Interfaces to Databases (NLIDB) systems eliminate the requirement for an end user to use complex query languages like SQL, by translating the input natural language (NL) queries to SQL automatically. Although a significant volume of research has focused on this space, most state-of-the-art systems can at best handle simple select-project-join queries. There has been little to no research on extending the capabilities of NLIDB systems to handle complex business intelligence (BI) queries that often involve nesting as well as aggregation. In this paper, we present ATHENA++, an end-to-end system that can answer such complex queries in natural language by translating them into nested SQL queries. In particular, ATHENA++ combines linguistic patterns from NL queries with deep domain reasoning using ontologies to enable nested query detection and generation. We also introduce a new benchmark data set (*FIBEN*), which consists of 300 NL queries, corresponding to 237 distinct complex SQL queries on a database with 152 tables, conforming to an ontology derived from standard financial ontologies (FIBO and FRO). We conducted extensive experiments comparing ATHENA++ with two state-of-the-art NLIDB systems, using both *FIBEN* and the prominent *Spider* benchmark. ATHENA++ consistently outperforms both systems across all benchmark data sets with a wide variety of complex queries, achieving 88.33% accuracy on *FIBEN* benchmark, and 78.89% accuracy on *Spider* benchmark, beating the best reported accuracy results on the dev set by 8%.

**PVLDB Reference Format:**
Jaydeep Sen, Chuan Lei, Abdul Quamar, Fatma Özcan, Vasilis Efthymiou, Ayushi Dalmia, Greg Stager, Ashish Mittal, Diptikalyan Saha, and Karthik Sankaranarayanan. ATHENA++: Natural Language Querying for Complex Business Intelligence Queries. *PVLDB*, 13(11): 2747-2759, 2020.
DOI: https://doi.org/10.14778/3407790.3407858

## 1. INTRODUCTION

Recent advances in natural language understanding and processing have fueled a renewed interest in Natural Language Interfaces to Databases (NLIDB) [22, 7, 26]. By 2022, 70% of white-collar workers are expected to interact with conversational systems on a daily basis[1]. The reason behind such increasing popularity of NLIDB systems is that they do not require the users to learn a complex query language such as SQL, to understand the exact schema of the data, or to know how the data is stored. NLIDB systems offer an intuitive way to explore complex data sets, beyond simple keyword-search queries.

Early NLIDB systems [6, 32] allowed only queries in the form of a set of keywords, which have limited expressive power. Since then, there have been works [20, 27, 31, 11, 29, 39, 34, 8] that interpret the semantics of a full-blown natural language (NL) query. Rule-based and machine learning-based approaches are commonly used to handle NL-to-SQL query translation. ATHENA [29] and NaLIR [20] are two representative rule-based systems that use a set of rules in producing an interpretation of the terms identified in the user NL query. Several machine learning-based approaches [11, 27, 31, 39, 34] have shown promising results in terms of robustness to NL variations, but these systems require large amounts of training data, which limits their re-usability in a new domain. Others require additional user feedback [18, 23] or query logs [8] to resolve ambiguities, which can be detrimental to user experience or can be noisy and hard to obtain. Most of these works generate simple SQL queries, including selection queries on a single table, aggregation queries on a single table involving `GROUP BY` and `ORDER BY`, and single-block SQL queries involving multiple tables (`JOIN`).

In this paper, we particularly focus on business intelligence (BI) and data warehouse (DW) queries. Enterprises rely on such analytical queries to derive crucial business insights from their databases, which contain many tables, and complex relationships. As a result, analytical queries on these databases are also complex, and often involve nesting and many SQL constructs. Although there have been some attempts [20, 10, 9] to generate such analytical queries (e.g., aggregation, join, or nested), to the best of our knowledge, none of these NLIDB systems can consistently provide high-quality results for NL queries with complex SQL constructs across different domains [7].

There are two major challenges for nested query handling in NLIDB: nested query detection, i.e., determining whether a nested query is needed, and nested query building, i.e., determining the sub-queries and join conditions that constitute the nested query.

**Nested Query Detection Challenge.** Detecting whether a NL query needs to be expressed as a nested (SQL) query is non-trivial

[1]Gartner - https://www.gartner.com/smarterwithgartner/chatbots-will-appeal-to-modern-workers/


*Proceedings of the VLDB Endowment,* Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407858

**Table 1: NL Query Examples and Nested Query Type**

| Query | NL Query Example | Type |
|---|---|---|
| $Q_1$ | Show me the customers who are also account managers. | Type-N |
| $Q_2$ | Show me Amazon customers who are also from Seattle. | Non-Nested |
| $Q_3$ | Who has bought more IBM stocks than they sold? | Type-JA |
| $Q_4$ | Who has bought and sold the same stock? | Type-J |
| $Q_5$ | Which stocks had the largest volume of trade today? | Type-A |
| $Q_6$ | Who has bought stocks in 2019 that have gone up in value? | Type-J |
| $Q_7$ | Show me all transactions with price more than IBM's average in 2019. | Type-JA |
| $Q_8$ | The number of companies having average revenues more than 1 billion last year. | Type-JA |

due to $(i)$ ambiguities in linguistic patterns, and $(ii)$ the variety in the types of nested queries.

*Ambiguities in linguistic patterns.* Consider the queries $Q_1$ and $Q_2$ from Table 1. Intuitively, $Q_1$ requires a nested query to find the intersection between customers and account managers, because the key phrase *"who are also"* indicates a potential nesting, and *"account managers"* and *"customers"* refer to two tables in the database schema. On the other hand, the same phrase *"who are also"* in $Q_2$ does not lead to a nested query, because *"Seattle"* is simply a filter for *"customers"*, not a table like *"account managers"*. In other words, linguistic patterns alone are not sufficient in detecting nested queries. Rather, we need to reason over the domain semantics in the context of the query to identify whether a nested query is needed or not.

*Variety in the types of nested queries.* Linguistic patterns may lead to different types of nested queries. Table 1 lists several examples of different nested query types. In this paper, we adapt the nested query classification from [17]. These nested query types will be further explained in Section 2.3. Query $Q_3$ indicates a comparison by the phrase *"more than"*, although the two words in this phrase are not contiguous in the NL query. Query $Q_4$ does not have an explicit comparison phrase, but still requires a nested query to enforce the *"same stock"*. In this case, the set of people used in the inner query references to the people in the outer query, creating a correlation between inner and outer query blocks. The key phrases to detect such nesting are *"everyone"* and *"same stock"*.

**Nested Query Building Challenge.** There are two challenges for nested query building: $(i)$ finding proper sub-queries (i.e., the outer and inner queries), and $(ii)$ identifying the correct join conditions between the outer and inner queries.

*Finding proper sub-queries.* Consider the query $Q_6$ from Table 1. The key phrase *"gone up"* indicates that $Q_6$ needs to be interpreted into a nested SQL query. If we naïvely use the position of phrase *"gone up"* to segregate $Q_6$ into outer and inner query tokens, the tokens *"stocks"* and *"value"* belong to the outer and the inner queries, respectively. However, the token *"value"* is also relevant to the outer query since it specifies the specific information associated with the stocks. Similarly, the token *"stocks"* is relevant to the inner query as well. Hence, segregating the tokens in a NL query for different sub-queries including the implicitly shared ones is critical to the correctness of the resulting nested query.

*Deriving join conditions.* Linguistic patterns in the NL queries often contain hints about join conditions between the outer and inner queries. However, deriving the correct join condition solely based on these patterns can be challenging. Consider the query $Q_6$ again, in which the phrase *"gone up"* indicates a comparison ($>$) operator. In addition, a linguistic dependency parsing identifies that two tokens *"stocks"* and *"value"* are dependent on the phrase *"gone up"*. It appears that the join condition would be a comparison between *"stocks"* and *"value"*. However, reasoning over the semantics of domain schema shows that only the token *"value"* refers to a numeric type, which is applicable to *"stocks"*. Hence, the correct join condition is a comparison between the *"value"* of both sub-queries. Clearly, deriving a correct join condition requires an NLIDB system to not only exploit linguistic patterns but also understand the semantics of domain schema.

In this paper, we present ATHENA++, an end-to-end NLIDB system that tackles the above challenges in generating complex nested SQL queries for analytics workloads. We extend ATHENA++ on our earlier work [29], which uses an ontology-driven two-step approach, but does not provide a comprehensive support for all nested query types. We use domain ontologies to capture the semantics of a domain and to provide a standard description of the domain for applications to use. Given an NL query, ATHENA++ exploits linguistic analysis and domain reasoning to detect that a nested SQL query needs to be generated. The given NL query is then partitioned into multiple *evidence sets* corresponding to individual sub-queries (inner and outer). Each evidence set for a query block is translated into queries expressed in Ontology Query Language (OQL), introduced in [29], over the domain ontology, and these OQL queries are connected by the proper join conditions to form a complete OQL query. Finally, the resulting OQL query is translated into a SQL query by using mappings between the ontology and database, and executed against the database.

**Contributions.** We highlight our main contributions as follows:

- We introduce ATHENA++, which extends ATHENA [29] to translate complex analytical queries expressed in natural language into *possibly* nested SQL queries.
- We propose a novel nested query detection method that combines linguistic analysis with deep domain reasoning to categorize a natural language query into four well-known nested SQL query types [17].
- We design an effective nested query building method that forms proper sub-queries and identifies correct join conditions between these sub-queries to generate the final nested SQL query.
- We provide a new benchmark (*FIBEN*[2]) which emulates a financial data warehouse with data from SEC [5] and TPoX [25] benchmark. *FIBEN* contains a large number of tables per schema compared to existing benchmarks, as well as a wide spectrum of query pairs (NL queries with their corresponding SQL queries), categorized into different nested query types.
- We conduct extensive experimental evaluations on four benchmark data sets including FIBEN, and the prominent *Spider* benchmark [37]. ATHENA++ achieves 88.33% accuracy on *FIBEN*, and 78.89% accuracy on *Spider* (evaluated on the dev set), outperforming the best reported accuracy on the dev set (70.6%) by 8%.[3]

The rest of the paper is organized as follows. Section 2 introduces the basic concepts, including the domain ontology, ontology-driven natural language interpretation, and the nested query types. Section 3 provides an overview of our ATHENA++ system. Sections 4 and 5 describe the nested query detection and translation, respectively. We provide our experimental results in Section 6, review related work in Section 7, and finally conclude in Section 8.

[2]Available at https://github.com/IBM/fiben-benchmark

[3]Based on the results published on `https://yale-lily.github.io/spider` (last accessed: 07/15/2020).

## 2. BACKGROUND

In this section, we provide a short description of domain ontologies, because we rely on them for domain reasoning. Then, we recap the ontology-driven approach of ATHENA [29], and describe four types of nested SQL queries that ATHENA++ targets.

### 2.1 Domain Ontology

We use domain ontologies to capture the semantics of the domain schema. A domain ontology provides a rich and expressive data model combined with a powerful object-oriented paradigm that captures a variety of real-world relationships between entities such as *inheritance*, *union*, and *functionality*. We use OWL [4] for domain ontologies, where real-world entities are captured as concepts, each concept has zero or more data properties, describing the concept, and zero or more object properties, capturing its relationships with other concepts. We consider three types of relationships: $(i)$ isA or inheritance relationships where all child instances inherit some or all properties of the parent concept, $(ii)$ union relationships where the children of the same parent are mutually exclusive and exhaustive, i.e., every instance of the parent is an instance of one of its children, and finally $(iii)$ functional relationships where two concepts are connected via some functional dependency, such as a listed security provided by a corporation. We represent an ontology as $O = (C, R, P)$, where $C$ is a set of concepts, $R$ is a set of relationships, and $P$ is a set of data properties. We use the term *ontology element* to refer to a concept, relationship, or property of an ontology. Figure 1 shows a snippet of the financial ontology FIBEN, corresponding to our new benchmark data set.

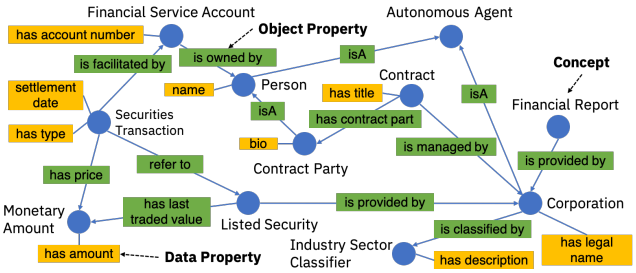


**Figure 1: Snippet of a Financial (FIBEN) Ontology**

### 2.2 Ontology-driven NL Query Interpretation

In this paper, we extend the ontology-driven approach introduced in [29] to handle more complex nested queries. Here, we provide a short description. The approach has two stages. In the first stage, we interpret an NL query against a domain ontology, and generate an intermediate query expressed in Ontology Query Language (OQL). OQL is specifically designed to allow expressing queries over a domain ontology, regardless of the underlying physical schema. In the second stage, we compile and translate the OQL query into SQL using ontology-to-database mappings. The ontology-to-physical-schema mappings are an essential part of query interpretation, and can be either provided by relational store designers or generated as part of ontology discovery [19].

Specifically, given an NL query, we parse it into *tokens* (i.e., sets of words), and annotate each token with one or more ontology elements, called *candidates*. Intuitively, each annotation provides evidence about how these candidates are referenced in the query. There are two types of matches. In the first case, the tokens in the query match to the names of concepts or the data properties directly. In the second case, we utilize a special semantic index, called Translation Index (TI), to match tokens to instance values in the data. For example, the token *"IBM"* in $Q_3$ from Table 1 can be mapped to the data properties *Corporation.name* and *ListedSecurity.hasLegalName* amongst other ontology elements in Figure 1. Formally, an *evidence* $v_i : t_i \mapsto E_i$ is a mapping of a token $t_i$ to a set of ontology elements $E_i \subseteq \{C \cup R \cup P\}$. The output of the annotation process is a set of evidences (each corresponding to a token in the query), which we call *Evidence Set (ES)*.

Next, we iterate over every evidence in $ES$ and select a single ontology element ($e_i \in E_i$) from each evidence's candidates ($E_i$) to create an *interpretation* of the given NL query. Since every token may have multiple candidates, the query may have multiple interpretations. Each interpretation is represented by an interpretation tree. An *interpretation tree* (hereafter called *ITree*), corresponding to one interpretation and an Ontology $O = (C, R, P)$, is formally defined as *ITree* = $(C', R', P')$ such that $C' \subseteq C$, $R' \subseteq R$, and $P' \subseteq P$. In order to select the optimal interpretation(s) for a given query, we rely on a Steiner Tree-based algorithm [29]. If the Steiner Tree-based algorithm detects more than one optimal solutions, then we end up with multiple interpretations for the same query. A unique OQL query is produced for each interpretation. Finally, each OQL query is translated into a SQL query by using a given mapping between the domain ontology and database schema.

### 2.3 Nested Query Types

For the following discussion, we assume that a SQL query has only one level of nesting, which consists of an outer block and an inner block. Further, it is assumed that the `WHERE` clause of the outer block contains only one nested predicate. These assumptions cause no loss of generality, as shown in [17].

**Table 2: Nested Query Types**

| Query Types | Aggregation | Correlation between Inner & Outer Queries | Division Predicate |
|---|---|---|---|
| **Type-A** | ✓ | ✗ | ✗ |
| **Type-N** | ✗ | ✗ | ✗ |
| **Type-J** | ✗ | ✓ | ✗ |
| **Type-JA** | ✓ | ✓ | ✗ |
| **Type-D** | ✗ | ✓ | ✓ |

Following the definitions in [17], we assume that a nested SQL query can be composed of five basic types of nesting (Table 2). In summary, Type-A queries do not contain a join predicate that references the relation of the outer query block, but contain an aggregation function in the inner block. Type-N queries contain neither a correlated join between an inner and outer block, nor an aggregation in the inner block. Type-J queries contain a join predicate that references the relation of the outer query block but no aggregation function in the inner block, and finally Type-JA queries contain both a correlated join as well as an aggregation. A join predicate and a division predicate together give rise to a Type-D nesting, if the join predicate in either inner query block (or both) references the relation of the outer block. Since the division operator used in Type-D queries does not have a direct implementation in relational databases [13], we choose not to translate NL queries into Type-D queries. Hence we focus on detecting and interpreting NL queries corresponding to the first four nesting types [13]. Table 1 lists a few NL queries with their associated nesting types.

## 3. SYSTEM OVERVIEW

Figure 2 illustrates the architecture of ATHENA++ , extended from ATHENA [29]. Similar to ATHENA, an NL query is first translated into OQL queries over the domain ontology, and then, each OQL query is translated into a SQL query by using the mappings between the ontology and the database, and executed against

the database. In this process, we re-use some of the components (in grey) introduced in [29], including *Translation Index, Domain Ontology, Ontology to Database Mapping*, and *Query Translator*. The newly added components for nested query handling are *Nested Query Detector* and *Nested Query Building*.

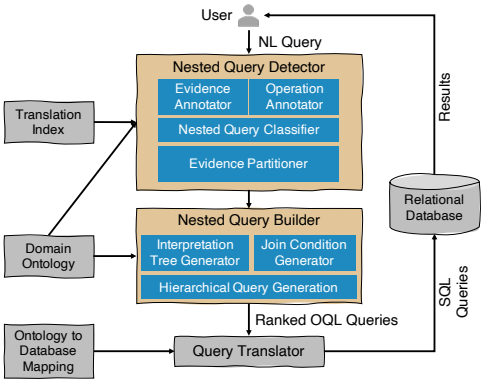


**Figure 2: System Architecture**

We use query $Q_6$ (*show me everyone who bought stocks in 2019 that have gone up in value*) from Table 1 as a running example to illustrate the workflow of ATHENA++ in Figure 3.

**Evidence Annotator** exploits Translation Index (TI) and Domain Ontology to map tokens in the query to data instances in the database, ontology elements, or SQL query clauses (e.g., `SELECT`, `FROM`, and `WHERE`). For example, the token *"stocks"* is mapped to *"ListedSecurity"* concept in the domain ontology. In addition, it also annotates the tokens as certain types such as time and number.

**Operation Annotator** leverages Stanford CoreNLP [24] for tokenization and annotating dependencies between tokens in the NL query. It also identifies linguistic patterns specific to nested queries, such as aggregation and comparison. The output of Evidence and Operation Annotators is an Evidence Set $ES$.

**Nested Query Classifier** takes as input the evidence set with annotations from Evidence and Operation Annotators, and identifies if the NL query corresponds to one of the nested query types of Table 2. In the example of Figure 3, it identifies that the phrase *"gone up"* refers to stock value which is compared between the outer and inner queries, and hence decides that $Q_6$ is a Type-J nested query.

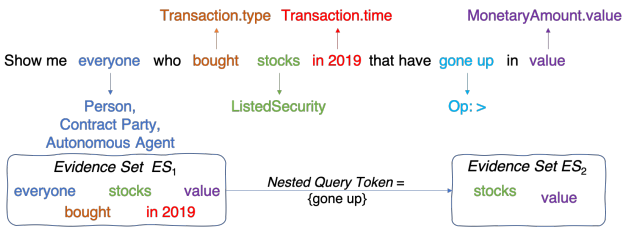


**Figure 3: Example Query ($Q_6$) and Evidence Sets**

**Evidence Partitioner** splits a given evidence set into potentially overlapping partitions by following a set of proposed heuristics based on linguistic patterns and domain reasoning, to delineate the inner and outer query blocks. As shown in Figure 3, for query $Q_6$ this results into two evidence sets $ES_1$ and $ES_2$ for the inner query and the outer query, respectively. $ES_1$ and $ES_2$ are connected by the detected nested query token *"gone up"*.

**Join Condition Generator** consumes a pair of evidence sets $ES_1$ and $ES_2$, and produces a join condition which can be represented by a comparison operator *Op* with two operands from $ES_1$ and $ES_2$, respectively. For example, the join condition for $Q_6$ is $ES_1.value > ES_2.value$.

**Interpretation Tree Generator** exploits the Steiner Tree-based algorithm introduced in [29] to return a single interpretation tree (*ITree*) for each evidence set produced by the Evidence Partitioner.

**Hierarchical Query Generation** is responsible for stitching the interpretation trees together by using the generated join conditions. In case of arbitrary levels of nesting, the Hierarchical Query Generation recursively builds the OQL query from the most inner query to the last outer query.

# 4. NESTED QUERY DETECTION

## 4.1 Evidence and Operation Annotators

As motivated in Section 1, the linguistic patterns and domain semantics are critical to the success of nested query detection. To discover such salient information, we first employ the open source Stanford CoreNLP [24] to tokenize and parse the input NL query. Then, for each token $t$, we introduce Operation Annotators to extract the linguistic patterns and Evidence Annotators to identify the domain semantics, respectively.

**Evidence Annotator.** The evidence annotator associates a token $t$ with one or more ontology elements including concepts, relationships, and data properties. To identify the ontology elements, we use Translation Index (TI) shown in Figure 2, which captures the domain vocabulary, providing data and metadata indexing for data values, and for concepts, properties, and relations, respectively. For example, in $Q_6$ the tokens *"stocks"* and *"bought"* are mapped to the concept *"ListedSecurity"* and the property *"Transaction.type"*, respectively, in the ontology shown in Figure 1. Alternative semantic similarity-based methods such as word embedding or edit distance can be utilized as well to increase matching recall, with a potential loss in precision.

The Evidence Annotator also annotates tokens that indicate time ranges (e.g., *"in 2019"* in $Q_6$) and then associates them with the ontology properties (e.g., *"Transaction.time"*) whose corresponding data type is time-related (e.g., Date). Similarly, the Evidence Annotator annotates tokens that mention numeric quantities, either in the form of numbers or in text, and subsequently matches them to ontology properties with numerical data types (e.g., Double). Finally, the Evidence Annotator further annotates certain identified Entity tokens that are specific to the `SELECT` clause of the outer SQL query, using POS tagging and dependency parsing from Stanford CoreNLP. Such entities are referred to as *Focus Entities*, as they represent what users want to see as the result of their NL queries. Table 3 lists several examples of the Evidence and Operation Annotators (separated by double lines). We also use other annotators for detecting various SQL query clauses such as `GROUPBY` and `ORDERBY`. These are orthogonal to nested query detection and translation, and hence we do not include them in Table 3.

**Operation Annotator.** The Operation Annotator assigns operation types to the tokens, when applicable. As shown in Table 3, our Operation Annotator primarily targets four linguistic patterns: count, aggregation, comparison, and negation. We distinguish count from other aggregation functions as it also applies to non-numeric data. A few representative examples of tokens corresponding to each annotation type are also presented in Table 3. Additionally, the Operation Annotator also leverages Stanford CoreNLP [24] for annotating dependencies between tokens in the NL query. The produced dependent tokens are then used in the nested query classification.

Note that each token can be associated with multiple annotations from both Evidence and Operation Annotators. For example, *"ev-*

*eryone"* in $Q_6$ is associated with the ontology concepts *"Person"*, *"Autonomous Agent"*, and *"Contract Party"*. The above annotations capture the linguistic patterns and domain semantics embedded in the given NL query and are utilized by the Nested Query Classifier to decide if the query belongs to one of the four nested query types described in Section 2.3.

**Table 3: Evidence & Operation Annotators**

| Annotations | Example Token $t$ |
|---|---|
| Entity | customer, stocks, etc. |
| Instance | IBM, California, etc. |
| Time | since 2010, in 2019, from 2010 to 2019, etc. |
| Numeric | 16.8, sixty eight, etc. |
| Measure | revenue, price, value, volume of trade, etc. |
| Count | count of, number of, how many, etc. |
| Aggregation | total/sum, max, min, average, etc. |
| Comparison | more/less than, gone up, etc.<br>equal, same, also, too, etc.<br>not equal, different, another, etc. |
| Negation | no, not, none, nothing, etc. |

## 4.2 Nested Query Classifier

Nested query detection uses annotated tokens from Evidence and Operation Annotators to detect if an input NL query is a nested query of a specific type. For each nested query type, the detection process uses a conjunction of rules based on (1) evidence and operation annotations, and (2) dependencies among specific annotated tokens. Intuitively, the rules on evidence and operation annotation check for the presence of certain linguistic patterns or specific ontology elements, and the rules on dependency analysis check if the annotated tokens are related to each other in a way corresponding to a specific nested query type. The nested query detection of ATHENA++ is presented in Algorithm 1.

The nested query detection algorithm first categorizes the operation annotations into two groups, *aggTokens* and *joinTokens*, respectively. The aggregation tokens indicate potential aggregation functions involved in the NL query, and the join tokens indicate potential join predicates between the inner and the outer query blocks. Note that there could be multi-level nesting in a given NL query. Hence, the tokens in both *aggTokens* and *joinTokens* groups are sorted in the order of their positions in the NL query. Below, we explain the detection rules for each nested query type in details, and use the queries in Table 1 as examples.

- *Type-N.* When *aggTokens* is empty, we examine whether the dependent tokens of each $jt$ in *joinTokens* indicate any correlation between the inner and outer queries. When the dependent tokens of $jt$ are a measure and a numeric value, intuitively it indicates that $jt$ is a numeric comparison, which does not require a join predicate referencing the outer query. Also, when the dependent tokens are of type Entity and the corresponding concepts in the ontology are siblings, it indicates that two entity sets in the inner and outer queries are directly comparable without a join.
  *Example.* In $Q_1$ ("*show me the customers who are also account managers*"), the token "*also*" refers to an equality between dependent entities "*customers*" and "*account managers*". Both entities are children of "*Person*" in the domain ontology, leading to a set comparison between the join results from inner and outer queries. Hence, both example queries are Type-N.

- *Type-A.* We consider a query as a candidate of Type-A if *joinTokens* is empty. The reason is that a Type-A nested query does not involve any correlation between inner and outer queries. Next, we check each token ($at$) in *aggTokens* to see if a dependent token of $at$ is of type Entity and also appears before $at$. The intuition is that an aggregation token is often applied to a previously mentioned entity in a query.
  *Example.* The token "*largest*" in $Q_5$ ("*Which stocks had the largest volume of trade today*") is of type Aggregation and is applied to the entity "*stock*". Hence, $Q_5$ is a Type-A nested query.

- *Type-J.* The prerequisite for Type-J is that a given NL query has at least one token $jt$ in *joinTokens*. For each $jt$, we find its dependent tokens $T_E$ of type Entity. Then, we further examine the operator annotation of $jt$. Specifically, if $jt$ is a negation and it appears before its dependent token in the query, then this indicates that the negation is applied to the entity in the inner query, and the outer query is correlated to the inner query by $jt$. If $jt$

**Algorithm 1:** Nested Query Detection Algorithm

**Input:** A natural language query $Q$
**Output:** Nested query tokens $nqTokens$

$nqTokens, aggTokens, joinTokens \leftarrow \emptyset$
$T \leftarrow tokenize(Q)$
$dep \leftarrow dependencyParsing(Q, T)$
**foreach** $t \in T$ **do**
  $t.annot \leftarrow annotators(t)$
  **if** $t.annot$ = *Count* ***or*** $t.annot$ = *Aggregation* **then**
    $aggTokens.add(t)$
  **else if** $t.annot$ = *Comparison* ***or*** *Negation* **then**
    $joinTokens.add(t)$
**if** $aggTokens = \emptyset$ ***and*** $joinTokens \neq \emptyset$ **then**
  **foreach** $jt \in joinTokens$ **do**
    $T_E \leftarrow dep.get(jt, Entity)$
    **if** $jt.annot \in$ {*Negation, Equality, Inequality*} **then**
      **if** $jt.next \in T_E$ ***and*** $jt.prev \in T_E$ ***and*** *sibling*$(jt.next, jt.prev)$ **then**
        $jt.nType \leftarrow$ Type-N
      **else if** $jt.next \in T_E$ ***or*** $jt.prev \in T_E$ **then**
        $jt.nType \leftarrow$ Type-J
    **if** $jt.annot$ = *Comparison* ***and*** $jt.prev.annot$ = *Measure* ***and*** $jt.next.annot$ = *Numeric* **then**
      $jt.nType \leftarrow$ Type-N
    $nqTokens.add(jt)$
**else if** $aggTokens \neq \emptyset$ ***and*** $joinTokens = \emptyset$ **then**
  **foreach** $at \in aggTokens$ **do**
    **if** $at.annot$ = *Aggregation* **then**
      **foreach** $dt \in dep.get(at, Entity)$ **do**
        **if** $at.pos > dt.pos$ **then**
          $at.nType \leftarrow$ Type-A
          $nqTokens.add(at)$
          **break**
**else if** $aggTokens \neq \emptyset$ ***and*** $joinTokens \neq \emptyset$ **then**
  **foreach** $jt \in joinTokens$ **do**
    **if** $jt.annot$ = *Comparison* ***and*** $jt.prev \in dep.get(jt, Measure)$ **then**
      $jt.nType \leftarrow$ Type-JA
      $nqTokens.add(jt)$
  **foreach** $at \in aggTokens$ **do**
    **if** $at.annot$ = *Aggregation* ***and*** $dep.get(at, Count) \neq \emptyset$ ***and*** $dep.get(at,$ *Entity*$) \neq \emptyset$ ***and*** $dep.get(at, Comparison) \neq \emptyset$ **then**
      $at.nType \leftarrow$ Type-JA
      $nqTokens.add(at)$
**return** $nqTokens$

is a comparison and its dependent entity is before or after in the query, then it is a strong signal that $jt$ compares the entities between the inner and outer queries.
*Example.* The token "*same*" in $Q_4$ ("*Who has bought and sold the same stock*") indicates a potential correlation between inner and outer queries, and it depends on the entity "*stock*". Therefore, $Q_4$ is a Type-J nested query.

- *Type-JA.* When neither *aggTokens* nor *joinTokens* are empty, the corresponding NL query is a candidate of Type-JA. For every $jt$ in *joinTokens*, we check if $jt$ is a comparison and the token before $jt$ is one of $jt$'s dependent tokens of type Measure. Intuitively, an aggregation is needed in the inner query so that the result of the inner query can be compared with the measure in the outer query. On the other hand, for every token $at$ in *aggTokens*, we check if $at$ is of type Aggregation and its dependent tokens are of type Count, Entity, or Comparison. Intuitively, the inner query with an aggregation is correlated with the outer query by a comparison, an entity reference, or another aggregation.
*Example.* The token "*more than*" in $Q_3$ ("*Who has bought more IBM stocks than they sold*") is of type Comparison, and the token "*stock*" implies the volume of trade (i.e., a measure). Consequently, $Q_3$ is a Type-JA nested query.

To support multi-level nesting, the nested query detection algorithm associates the detected nested query types to their corresponding tokens. This way, the appropriate evidence partitioning strategies (Section 4.3) can be applied to each nesting level depending on the associated type. If none of the tokens have a nested query type detected, then the given NL query is not a nested query.

## 4.3 Evidence Partitioner

A nested SQL query can be viewed as a hierarchy of sub-queries, where the results of the sub-queries are linked with the appropriate nested predicates on the sub-query results. Without loss of generality, we assume that an NL query is roughly split into two individual sub-queries, which are referred to as an *outer query* and an *inner query*. If a query requires multi-level nesting, the inner query can be further split into outer and inner sub-queries. These sub-queries are connected with the corresponding nested predicate as well.

The Evidence Partitioner is designed to split the evidence set $ES$ corresponding to the complete query $Q$ into two subsets $ES_1$ and $ES_2$, which contain evidence related to the outer query and the inner query of $Q$, respectively. A straightforward approach would be to partition $ES$ based on the nested query tokens $nqTokens$ from Algorithm 1. The tokens that appear before (after) a nested query token would belong to the outer (inner) query evidence set $ES_1$ ($ES_2$). However, such partitioning often fails to capture sufficient information (evidence) for each sub-query, resulting in an incorrect SQL query or even failing to produce one.

*Example.* The nested query token of $Q_6$ is *"gone up"*. If we only consider the tokens after the nested query token, the inner query evidence set $ES_2$ would be {*"value"*} that is insufficient to produce a valid nested SQL query. In fact, from the detected linguistic patterns, we know that $Q_6$ is a nested query of Type-J, where a join predicate exists between the outer and inner query blocks. We also identify that the token *"stocks"* has a dependency with *"value"*. Hence, *"stocks"* should be part of $ES_2$ as well.

To overcome such shortcomings of the aforementioned straightforward approach, we discover a set of partitioning heuristics based on the detected nested query types associated with the nested query tokens, as well as the linguistic patterns and domain semantics associated with all tokens from the given query. Driven by these heuristics, our Evidence Partitioner generates both outer ($ES_1$) and inner ($ES_2$) evidence sets appropriately.

- *Heuristic 1 (Co-reference).* In Type-J and Type-JA queries, a join predicate references the outer query block. Hence, any token $t$ in $ES_1$, co-referenced by a token $t'$ in $ES_2$, should be part of $ES_2$. We utilize Stanford CoreNLP [24] for co-reference resolution.
*Example.* In $Q_3$ (Table 1), the token *"who"* in $ES_1$ is added to $ES_2$, since *"they"* in $ES_2$ refers to *"who"*.
- *Heuristic 2 (Time sharing).* Any token $t$ of type Time should be part of both $ES_1$ and $ES_2$ unless $ES_1$ and $ES_2$ contain separate tokens of type Time of their own. The time sharing heuristic is generic to all nested query types, since it aims to discover the hidden time predicate of $ES_1$ or $ES_2$ from the NL query.
*Example.* The token *"in 2019"* in $Q_6$ should be added to $ES_2$ as it implicitly specifies the time range for the inner query.
- *Heuristic 3 (Instance sharing).* Any token $t$ of type Instance in either $ES_1$ or $ES_2$ should be part of both $ES_1$ and $ES_2$ for non-aggregation queries. We introduce the instance sharing heuristic for Type-J and Type-N nested queries since the instance is often used as part of a predicate in a sub-query.
*Example.* The token *"IBM"* in a slight variation of $Q_6$ (*"who has bought IBM stock in 2019 that has gone up in value"*) is a data instance, and it should be added to $ES_2$ since the inner query is about the value of *"IBM"*.
- *Heuristic 4 (Focus sharing).* If a non-numeric comparison is detected but the inner query does not contain a Focus Entity, then the outer query shares its Focus Entity with the inner query in order to complete the comparison. Namely, the Focus Entity in $ES_1$ should be shared with $ES_2$.
*Example.* In $Q_4$ (Table 1), the Focus Entity *"who"* of $ES_1$ is shared with $ES_2$.
- *Heuristic 5 (Argument sharing).* If one of the arguments of a comparison is missing from either $ES_1$ or $ES_2$, then the available argument should be shared between $ES_1$ and $ES_2$. Similar to the time sharing heuristic, the argument sharing heuristic is also generic to all nested query types. The reasons are twofold: a comparison always requires two arguments, and the arguments can be associated with or without an aggregation.
*Example.* In $Q_4$, *"stocks"* should be shared since both *"bought"* and *"sold"* have the same argument. Similarly, in $Q_7$, *"price"* should be shared with the inner block *"IBM's average in 2019"*.
- *Heuristic 6 (Entity/Instance sharing in numeric comparison).* Every token $t$ of Entity type in $ES_1$ should be part of $ES_2$, if $t$ has a dependent entity in $ES_2$; or every token $t$ of Instance type in $ES_1$ should be part of $ES_2$, if $t$ has a functional relationship with the comparison argument.
*Example.* In $Q_6$, the token *"stocks"* has a dependency with the token *"value"* in $ES_2$, so it is added to $ES_2$. In $Q_3$, the token *"IBM"* (i.e., an instance of Corporation.name) has a functional relationship in the domain ontology with the concept *"stocks"*. In this case, *"IBM"* is shared with $ES_2$ as well.

In Table 4, we summarize the applicability of each partitioning heuristic with respect to all nested query types. Note that the above heuristics are not mutually exclusive, as they only introduce evidence to either $ES_1$ or $ES_2$ without removing any evidence. Consequently, our evidence partitioning algorithm (Algorithm 2) recursively utilizes these heuristics at each level of nesting as long as they are applicable to the corresponding nested query type.

The design goal of the above partitioning heuristics is to discover latent linguistic information for each sub-query in the given NL query. We observe that these heuristics work well in practice,

**Table 4: Partitioning Heuristics w.r.t Nested Query Type**

| | Type-N | Type-A | Type-J | Type-JA |
|---|---|---|---|---|
| Heuristic 1 | | | ✓ | ✓ |
| Heuristic 2 | ✓ | ✓ | ✓ | ✓ |
| Heuristic 3 | ✓ | | ✓ | |
| Heuristic 4 | ✓ | | ✓ | |
| Heuristic 5 | ✓ | ✓ | ✓ | ✓ |
| Heuristic 6 | | ✓ | ✓ | ✓ |

**Algorithm 2:** Evidence Partitioning Algorithm

**Input:** Nested query tokens $nqTokens$, Evidence set $ES$
**Output:** A list of evidence sets $List_{ES}$

1 $List_{ES} \leftarrow \emptyset$
2 **foreach** $nqt \in nqTokens$ **do**
3 $ES_1 \leftarrow ES.before(nqt)$
4 $ES_2 \leftarrow ES.after(nqt)$
5 **if** $nqt.type$ = *Type-N* **then**
6 $applyHeuristic(ES_1, ES_2, H_2, H_3, H_4, H_5)$
7 **else if** $nqt.type$ = *Type-A* **then**
8 $applyHeuristic(ES_1, ES_2, H_5, H_6)$
9 **else if** $nqt.type$ = *Type-J* **then**
10 $applyHeuristic(ES_1, ES_2, H_1, H_2, H_3, H_4, H_5, H_6)$
11 **else if** $nqt.type$ = *Type-JA* **then**
12 $applyHeuristic(ES_1, ES_2, H_1, H_2, H_5, H_6)$
13 $List_{ES}.add(ES_1)$
14 $ES \leftarrow ES_2$
15 $List_{ES}.add(ES)$
16 **return** $List_{ES}$

however there are cases when these heuristics lead to erroneous inferences or fail to infer critical information from the query, due to the ambiguous nature of linguistic patterns. For example, Heuristic 2 is designed to discover the hidden time predicate from a sub-query. However, the true intent of $Q_7$ in Table 1 is to retrieve all transactions in history, then applying Heuristic 2 would incorrectly infer 2019 as the time predicate for all transactions. A more detailed error analysis in Section 6.2 provides more insights into the effectiveness of the proposed partitioning heuristics. Our heuristics are shown to be effective and robust to different nested query types across a variety of domain-specific data sets in our experimental evaluation (Section 6).

# 5. NESTED QUERY BUILDING

## 5.1 Join Condition Generator

The Evidence Partitioner (Algorithm 2) splits the evidence set $ES$ for a given NL query into a list of potentially overlapping evidence sets $ES_1, ES_2, \ldots, ES_n$. Intuitively, each evidence set represents a different sub-query, and two adjacent evidence sets with nested query tokens (resulting from Algorithm 1) determine the *join conditions* between those sub-queries. Formally, a join condition $jc$ = ($ES_i.c_j$, *op*, $ES_{i+1}.c_k$) refers to the evidence $c_j$ and $c_k$ in two adjacent evidence sets $ES_i$ and $ES_{i+1}$, respectively, and to the join operator *op* that connects them.

Our Join Condition Generator first decides the operator to use for the join condition between two adjacent evidence sets based on the nested query token. In the example of $Q_6$ (Figure 3), the operator *op* between $ES_1$ and $ES_2$ is *"greater than"* ($>$), which is extracted from the operation annotation of the nested query token *"gone up"*. As listed in Table 3, we support a variety of commonly used operators, including equal ($=$), greater than ($>$), less than ($<$), not-equal ($<>$), and `IN` to form the join condition.

Next, the Join Condition Generator needs to identify the join operands associated with the operator. The join operands are selected among the elements of the two adjacent evidence sets to be joined. To identify the correct evidences as join operands, the Join Condition Generator relies on information from the annotations associated with each token. Specifically, we leverage the nested query token dependencies to identify its dependent tokens and use them as the join operands. In case a join operand cannot be identified from one of the evidence sets ($ES_1$ or $ES_2$), we share the join operand identified from the other evidence set. It often happens when the nested query token is a numeric comparison.

In certain cases, there can be more than one join conditions between $ES_1$ and $ES_2$, derived from the nested query token dependencies. We consequently leverage the evidence and operation annotations of these tokens to choose the correct join condition. Specifically, if the join operator is a numeric comparison, then two join operands of this join operator have to be numeric as well. If the join operands of a join operator are two lists, then these two lists should be of the same entity type, or the entity types are siblings in the domain ontology (i.e., two children of the same parent concept or two members of the same union concept). This validation is critical to exclude certain 'bad' queries such as *"find all customers who are corporations"*, since *"customers"* and *"corporations"* are not comparable from a semantic perspective. The overall join condition generation method is presented in Algorithm 3.

Algorithm 3 takes as inputs a list of nested query tokens and a list of evidence sets produced by Algorithm 2. For each nested query token $nqt$, it first decides the join operator based on $nqt$, finds the dependent tokens of $nqt$, and retrieves the evidence sets $ES_1$ and $ES_2$ immediately before and after $nqt$ (Lines 2-6). Then, for each dependent token $dt$ of $nqt$, the algorithm checks if its position in the NL query is before (after) $nqt$ and if it also belongs to $ES_1$ ($ES_2$). If so, the dependent token is added to the join evidence set $JE_1$ ($JE_2$) accordingly (Lines 7-11). For example, the nested query tokens *"who are also"* of $Q_1$ in Table 1 have two dependent tokens *"customers"* and *"account managers"* in the outer ($ES_1$) and inner ($ES_2$) evidence sets, respectively. Hence, *"customers"* and *"account managers"* are added to $JE_1$ and $JE_2$, respectively.

The algorithm then checks the join operator type. If the operator is of type numeric, the algorithm selects the tokens that are of numeric type from $JE_1$ and $JE_2$. If the join evidence set does not contain any token of numeric type, then the token introduced by Heuristic 5 (in Section 4.3) is chosen from $JE_1$ and $JE_2$ (Lines 12-14). This is because Heuristic 5 is designed to share the argument of a comparison. For example, in $Q_6$ (Table 1), the only dependent token of the nested query token *"gone up"* ($>$) is *"value"* in $JE_2$. Hence, the token *"value"* shared by Heuristic 5 is added to $JE_1$ to form the join condition $jc$ = ($ES_1$.*value*, $>$, $ES_2$.*value*), since the comparison ($>$) is between the *"value"* of *"stocks"*.

If the operator is of type list, the algorithm keeps the tokens with the same entity types in both $JE_1$ and $JE_2$. If multiple tokens remain, then the token introduced by Heuristic 4 (Section 4.3) is kept (Lines 15-16), since Heuristic 4 aims at sharing a Focus Entity for a non-numeric comparison (i.e., list). Intuitively, the shared token *"who"* of $Q_4$ (Table 1) refers to the same entity (i.e., people) in both inner and outer queries. Consequently, the join condition is $jc$ = ($ES_1$.*who*, `IN`, $ES_2$.*who*).

Thereafter, if $JE_1$ and $JE_2$ are not empty, then the join condition between $ES_1$ and $ES_2$ is ($JE_1$, *op*, $JE_2$) (Lines 17-18). Otherwise, the join operand in $ES_1$ (or $ES_2$) is shared with $ES_2$ (or $ES_1$) (Lines 19-22). Finally, the list of join conditions between

all adjacent evidence sets is returned (Line 24). These join conditions are returned in the order of their appearance in the given query, from the outermost query to the innermost query.

**Algorithm 3:** Join Condition Generation Algorithm

**Input:** Nested query tokens $nqTokens$, A list of evidence sets $List_{ES}$
**Output:** A list of join conditions $List_{JC}$

1 $List_{JC}, JE_1, JE_2 \leftarrow \emptyset$
2 **foreach** $nqt \in nqTokens$ **do**
3 $op \leftarrow map(nqt)$
4 $dTokens \leftarrow dep.get(nqt)$
5 $ES_1 \leftarrow List_{ES}.before(nqt)$
6 $ES_2 \leftarrow List_{ES}.after(nqt)$
7 **foreach** $dt \in dTokens$ **do**
8 **if** $dt.pos < nqt.pos$ ***and*** $dt \in ES_1$ **then**
9 $JE_1.add(dt)$
10 **else if** $dt.pos > nqt.pos$ ***and*** $dt \in ES_2$ **then**
11 $JE_2.add(dt)$
12 **if** $op.type = Numeric$ **then**
13 $JE_1 \leftarrow JE_1.select(Numeric, H_5)$
14 $JE_2 \leftarrow JE_2.select(Numeric, H_5)$
15 **else if** $op.type = List$ **then**
16 $JE_1, JE_2 \leftarrow intersectEntity(JE_1, JE_2, H_4)$
17 **if** $JE_1 \neq \emptyset$ ***and*** $JE_2 \neq \emptyset$ **then**
18 $jc \leftarrow (JE_1, op, JE_2)$
19 **else if** $JE_1 \neq \emptyset$ **then**
20 $jc \leftarrow (JE_1, op, JE_1)$
21 **else**
22 $jc \leftarrow (JE_2, op, JE_2)$
23 $List_{JC}.add(jc)$
24 **return** $List_{JC}$

## 5.2 Interpretation Tree Generator

The interpretation tree generation process exploits the Steiner Tree-based algorithm introduced in [29] to return a ranked list of interpretations for each evidence set produced by the Evidence Partitioner. These interpretations are ranked based on the number of edges (i.e., relations) contained in each interpretation ($ITree$). In case there are multiple top-ranked interpretations with the same compactness (i.e., the same number of edges present in the interpretation trees) for an evidence set, all such interpretations are kept. The top-ranked $ITrees$ for $ES_1$ ($ITree_1$) and $ES_2$ ($ITree_2$) of $Q_6$ are shown in Figure 4. For simplicity, we assume that only one $ITree$ represents each evidence set. We refer to [29] for further details on handling multiple $ITrees$.

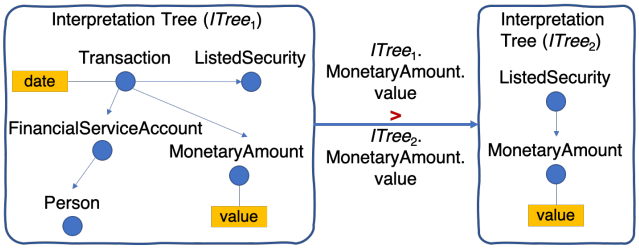


**Figure 4: Hierarchy of Interpretation Trees for $Q_6$**

## 5.3 Hierarchical Query Generator

The Hierarchical Query Generator is responsible for generating a complete OQL query by connecting the individual *ITrees* (Section 5.2) with the corresponding join conditions (Section 5.1). This process is described in Algorithm 4.

As described earlier, each evidence set, generated from evidence partitioning, is represented by an *ITree*. Following the order of the evidence sets, the corresponding interpretation trees of these evidence sets also form an ordered list of *ITrees*. Conceptually, two adjacent *ITrees* can be connected through the same join condition as their corresponding evidence sets. Their connection can be represented as an edge linking the two *ITrees*, labeled with the join condition between the identified join operands.

Depending on the join condition, an *ITree* of an inner query can be either used in a `WHERE` (Lines 14-15) or a `HAVING` (Lines 12-13) clause of the outer query's *ITree*. In addition to the join condition identified in Section 5.1, the identical evidences between the inner and outer evidence sets (Line 7) which are not join arguments are asserted to be the same in the hierarchical query building. Such evidence correlates the inner and outer queries, and hence is used as part of a `WHERE` clause as well (Lines 8-10). In case of multi-level nesting, the algorithm starts from the innermost query and iteratively stitches the newly built outer query to the existing inner query, until reaching the last outer query (Lines 16-17). Figure 4 depicts a complete hierarchical interpretation tree of $Q_6$, while the corresponding OQL query of $Q_6$ is shown in the top of Figure 5.

**Algorithm 4:** Hierarchical Query Building Algorithm

**Input:** A list of interpretation trees $ITrees$, A list of join conditions $List_{JC}$, A list of evidence sets $List_{ES}$
**Output:** An OQL query *oqlQuery*

1 $oqlQuery, inner \leftarrow \emptyset$
2 **for** $i \leftarrow List_{JC}.size$ **to** 1 **do**
3 $jc \leftarrow List_{JC}.get(i)$
4 `WHERE`, `HAVING`, $JC', outer \leftarrow \emptyset$
5 $ES_1 \leftarrow List_{ES}.before(jc)$
6 $ES_2 \leftarrow List_{ES}.after(jc)$
7 $ES' \leftarrow (ES_1 \cap ES_2) \setminus jc.getArgs()$
8 **foreach** $e' \in ES'$ **do**
9 `WHERE`$.add((e', =, e'))$
10 $inner \leftarrow buildOQL(ES_2, ITrees.after(jc),$ `WHERE`$)$
11 `WHERE`$\leftarrow \emptyset$
12 **if** $jc.get(op).type = Numeric$ ***and*** $jc.getArg1().type = Aggregation$ **then**
13 $outer \leftarrow buildOQL(ES_1, ITrees.before(jc),$ `HAVING`$.add(jc))$
14 **else**
15 $outer \leftarrow buildOQL(ES_1, ITrees.before(jc),$ `WHERE`$.add(jc))$
16 $oqlQuery.addBefore(inner)$
17 $inner \leftarrow outer$
18 **return** *oqlQuery*

**Discussion.** As described above, the Nested Query Builder takes a bottom-up approach to create individual *Itrees* for each evidence set, and then build a complete OQL query based on these *Itrees*. Below, we highlight the observations that lead to this approach, which are also confirmed by our experimental evaluation (Section 6).

*Observation 1.* Two connected *ITrees* may contain some common nodes. However, it does not necessarily mean that these common nodes refer to the same objects in the target query. For example, both $ITree_1$ and $ITree_2$ contain *"ListedSecurity"* and *"MonetaryAmount"* in Figure 4. Considering the natural language query "show me everyone who bought stocks in 2019 that have gone up in value", it is evident the query is asking for the same *"stock"* across both outer and inner queries. Thus the node *"ListedSecurity"* corresponding to the natural language token *"stock"* indeed means the same object across two sub queries. However, it is not true for the other common node *"MonetaryAmount"*, which corresponds to the

natural language token *"value"*. In the outer query, the *"value"* corresponds to the bought (transaction) value of the stock, whereas it is the last traded value of the stock in the inner query. Therefore, the node *"MonetaryAmount"* should have two different objects in the target queries, even though it is common between two *ITrees*.

*Observation 2.* Even though two connected *ITrees* may contain common nodes, they still may not share the same set of edges between those common nodes. As shown in Figure 4, even though both *ITrees* contain *"ListedSecurity"* and *"MonetaryAmount"*, the edges between them are not the same in their corresponding *ITrees*. This intuitively follows *Observation 1*. For $ITree_1$ (i.e., the outer query), *"ListedSecurity"* is connected to *"MonetaryAmount"* via *"Transaction"* as the outer query is about purchase value of stocks. For $ITree_2$ (i.e., the inner query), *"ListedSecurity"* is connected to *"MonetaryAmount"* via *"hasLastTradedValue"* edge as the inner query is about the current value of the stock.

*Observation 3.* Two connected *ITrees* may share nothing in common, even when there is a join condition between them. Consider the query $Q_1$ in Table 1, where $ITree_1$={Customer} and $ITree_2$={Account Manager} do not share any common nodes, but a join condition $jc$ = ($ITree_1$.*Customer*, IN, $ITree_2$.*Account Manager*) exists between them. This is due to the fact that *"Customer"* and *"Account Manager"* are both children of the same parent concept *"Person"* in the domain ontology.

In summary, each *ITree* should have its own interpretation corresponding to each evidence set, since two *ITrees* cannot assume the same interpretation from identical nodes and edges as they may contain different context information. Hence, we build the nested OQL query using a bottom-up approach to ensure that each evidence is used in the right context (i.e., inner and/or outer queries).

## 5.4 Query Translation

We now present an overview of our Query Translator adopted from [29]. The Query Translator takes an OQL query as input, consisting of either a simple non-nested query or a nested query with an inner (or nested) query and outer query[4]. Since a nested OQL query can be a union of individual OQL queries, the algorithm maintains a set of OQL queries that need to be translated into equivalent SQL queries. To generate the SQL query, the query translator requires appropriate schema mappings that map concepts and relations represented in the domain ontology to appropriate schema objects in the target database, which also includes the relations between the concepts in the ontology and the primary and foreign key (PK-FK) constraints between the tables that correspond to these concepts. The PF-FK constraints and the mappings are used to construct a *join graph*, which is utilized to generate the appropriate join conditions for the nested SQL query.

The inner query of a nested query is supported in FROM, WHERE, and HAVING clauses of the outer query. While processing each of these clauses in the outer query, the Query Translator detects the presence of an inner (nested) OQL query and recursively processes the inner query to generate a corresponding SQL query that is then placed in the appropriate outer query clause within braces. In Figure 5, we show the nested SQL query generated by the Query Translator for an OQL query corresponding to $Q_6$ in Table 1. In this case, the inner query computes the maximum monetary amount for the listed security, which is then compared against the monetary amount in the outer query using the WHERE clause.

In this process, the Query Translator also handles many-to-many (*M:N*) and inheritance relationships in a special way. For each *M:N* relationship detected by the translator, an intermediate table is introduced as the two tables corresponding to the concepts of the *M:N* relationship cannot be joined directly. We use the intermediate table to break the original join into two inner joins through the PK-FK constraint. For inheritance relationships, the Query Translator introduces an additional join condition between the parent and child concepts, if the given OQL query only includes a child concept and references to a property from its parent concept. The additional join condition ensures that the connection to the originally referenced child concept is maintained. We refer to [29] for further details.

```
NL Query: Who bought stocks in 2019 that have gone up in value
OQL Query:
SELECT LS.hasLegalName, MA.hasAmount, P.Name
FROM   SecuritiesTransaction ST, Person P,
       ListedSecurity LS, MonetaryAmount MA
WHERE  ST.hasType = '1' AND MA.hasAmount <
       (SELECT Max(InnerMA.hasAmount)
        FROM  ListedSecurity InnerLS,
              MonetaryAmount InnerMA
        WHERE LS.id = InnerLS.id AND
              InnerLS->hasLastTradedValue=InnerMA) AND
       ST.hasSettlementDate >= '2019-01-01' AND
       ST.hasSettlementDate <= '2019-12-31' AND
       ST->isFacilitatedBy->isOwnedBy=oPerson AND
       ST->refersTo=LS AND
       ST->hasPrice=MA
SQL Query:
SELECT LS.hasLegalName, MA.hasAmount, P.hasPersonName
FROM FinancialServiceAccount FSA1
INNER JOIN SecuritiesTransaction ST
ON FSA1.financialserviceaccountId=ST.isFacilitatedBy
INNER JOIN Person P
ON FSA1.isOwnedBy=P.personId
INNER JOIN ListedSecurity LS
ON ST.refersTo=LS.listedsecurityId
INNER JOIN MonetaryAmount MA
ON ST.hasPrice=MA.monetaryamountId
WHERE ST.hasType = '1' AND
      MA.hasAmount < (SELECT Max(InnerMA.hasAmount)
      FROM ListedSecurity InnerLS
      INNER JOIN MonetaryAmount InnerMA
      ON InnerLS.hasLastTradedValue=InnerMA.monetaryamountId
      WHERE LS.ListedSecurityID=InnerLS.ListedSecurityID) AND
      ST.hasSettlementDate >= '2019-01-01' AND
      ST.hasSettlementDate <= '2019-12-31'
```

**Figure 5: SQL Generated for $Q_6$**

# 6. EXPERIMENTAL EVALUATION

## 6.1 Experimental Setup

**Infrastructure**. We implemented ATHENA++ in Java with JDK 1.8.0_45. All data sets are stored in IBM Db2®. We ran the ATHENA++ and two baseline systems on RedHat 7.7 with 32-core 2.0GHz CPU and 250GB of RAM.

**Data Sets and Workloads.** We evaluate the effectiveness of ATHENA++ in handling complex nested queries, using four benchmark data sets.

- *Microsoft Academic Search (MAS)* data set [3] contains bibliographic information for academic papers, authors, conferences, and universities. *MAS* includes 250K authors and 2.5M papers, and a set of 196 queries from [20]. We expanded the query set with 77 additional queries, expressed in both NL and SQL.
- *GEO* data set [38] contains geographical data about the United States (Geobase), as well as a collection of 250 NL queries. These NL queries are mapped to the corresponding SQL queries in [14]. We expanded the query set with 74 additional queries.
- *Spider* [37] is a state-of-the-art, large-scale data set for complex and cross-domain text-to-SQL tasks. It has evolved as one of the de facto benchmarks in the research community for testing the accuracy of NLIDB systems. It consists of multiple schemas, each with multiple tables. The queries in *Spider* cover a wide spectrum of complex SQL queries including different

[4]The language grammar supports an arbitrary level of nesting wherein each inner query can either be a simple or a nested query.

**Table 5: Data Sets, Ontologies, and Queries**

| Data Set | $\|C\|$ | $\|P\|$ | $\|R_F\|$ | #Tables/ schema | Total #queries | #Nested | #Aggregation (in/out/both) | #Negation | ORDERBY | GROUPBY | HAVING |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *MAS* | 17 | 23 | 20 | 15 | 273 | 74 | 100 | 0 | 25 | 43 | 21 |
| *GEO* | 18 | 16 | 28 | 6 | 324 | 70 | 153 | 15 | 6 | 117 | 9 |
| *Spider* | 4.1 | 25.9 | 3.3 | 4.1 | 1,034 | 161 | 578 | 46 | 241 | 277 | 79 |
| *FIBEN* | 152 | 664 | 159 | 152 | 300 | 170 | 192 | 18 | 26 | 112 | 78 |

clauses such as joining and nested query. Unlike the end-to-end deep-learning frameworks that rely on different training, development, and test sets, we evaluate ATHENA++ on the development set of *Spider*, which consists of 20 different schemas and 1,034 queries. The variety of domains and queries present a non-trivial challenge to the precision and recall of the systems, including ATHENA++ and other NLIDBs.

- *FIBEN* is a new benchmark built on top of a financial data set introduced in [30]. The *FIBEN* data set is a combination of two financial data sets, *SEC* [5] and *TPoX* [25]. The number of tables per database schema is at least an order of magnitude greater than the other benchmarks, resembling the complexity of an actual financial data warehouse. *FIBEN* ontology is a combination of FIBO [1] and FRO [2] ontologies, providing sufficient domain complexity to express real-world financial BI queries. FIBEN contains 300 NL queries, with 237 distinct SQL queries, corresponding to different nested query types. These NL queries are typical analytical queries generated by BI experts. We asked these experts to create NL queries, while making sure that the resulting SQL queries have a substantial coverage of all four nested query types, with a variety of SQL query constructs. The benchmark queries are available at `https://github.com/IBM/fiben-benchmark`.

Table 5 provides the detailed statistics about these data sets and the corresponding ontologies (in terms of the average number of concepts $|C|$, properties $|P|$ and relationships $|R_F|$) that describe the domain schema of these data sets. It also provides detailed information about the query workload for each data set used in our experimental evaluation, such as the number of nested queries (which are included in the total count of queries), the number of nested queries with aggregation (in the outer query or the inner query or both), the number of negation queries, etc. Table 6 further lists the number of queries for each nested query type in each data set, which enable us to evaluate NLIDB systems performance on different types of nested queries.

**Table 6: Number of Queries per Nested Query Type**

| Data Set | Type-N | Type-A | Type-J | Type-JA |
|---|---|---|---|---|
| *MAS* | 7 | 48 | 6 | 13 |
| *GEO* | 21 | 25 | 14 | 10 |
| *Spider* | 136 | 25 | 0 | 0 |
| *FIBEN* | 28 | 64 | 40 | 38 |

**Compared Systems.** We compare ATHENA++ to two state-of-the-art NLIDB systems, our earlier system ATHENA [29, 19] and NaLIR [20, 21].

- ATHENA++ and ATHENA both are rule-based NLIDB systems. We use an identical experimental setting for both ATHENA and ATHENA++ . Namely, we follow the same approach described in [15] to create an ontology corresponding to the schema of all four data sets, and instantiate the system thereafter.
- NaLIR is another state-of-the-art rule-based NLIDB. We evaluate the system in which the interactive communicator is disabled, because its application of user interaction is orthogonal to our approach. Hence NaLIR forms the SQL query based on the dependency parse tree structure alone.

**Metrics and Methodologies.** We evaluate ATHENA++ and the other two systems using the above described benchmark data sets and the associated query workloads. To measure the effectiveness of these systems, we use the following metrics.

- *Accuracy.* The accuracy is defined as the number of correctly generated SQL queries over the number of NL queries asked. These queries include both nested and non-nested queries. We define the *correctness* of a generated SQL query by comparing the query results generated by executing the ground truth SQL queries, and the ones generated by the SQL queries from the NLIDB systems. Note that accuracy may consider a generated SQL query as correct, even if it is syntactically different from the ground truth query but returns the same results. We report the accuracy of the evaluated methods over all queries in each benchmark data set (*Overall Accuracy*), as well as the accuracy when considering only the nested queries of each data set (*Nested Query Accuracy*).
- *Precision, Recall, and F1-score.* To verify the effectiveness and robustness of ATHENA++'s nested query detection and generation, we also measure the *precision*, *recall* and *F1-score* of ATHENA++ specifically for nested queries in each data set.
  *Nested Query Detector Precision.* The number of NL queries **correctly** classified as nested queries over the number of NL queries classified as nested queries.
  *Nested Query Detector Recall.* The number of NL queries **correctly** classified as nested queries over the number of NL queries that correspond to nested queries in the ground truth.
  *Nested Query Builder Precision.* The number of **correct** nested SQL generated over the number of nested SQL generated, among the correctly classified NL queries.
  *Nested Query Builder Recall.* The number of **correct** nested SQL generated over the number of nested SQL provided by the ground truth, among the correctly classified NL queries.
- *Performance Evaluation.* We also analyze ATHENA++'s SQL generation time, i.e., the time spent in translating the input NL query to the output SQL query. We run each query three times and report the average.

## 6.2 Experimental Results

### 6.2.1 Accuracy

Table 7 shows the overall accuracy of each system on the four benchmark data sets for their corresponding workload. In general, ATHENA++ outperforms NaLIR and ATHENA significantly across all data sets. In particular, ATHENA++ achieves 88.33% accuracy on the new *FIBEN* data set, which is 40% and 68% higher than ATHENA and NaLIR, respectively. This confirms that ATHENA++ is capable of handling complex analytics queries. It is also important to point out that ATHENA++ achieves 78.89% accuracy on the prominent *Spider* benchmark, outperforming the best reported accuracy of 65% by 13%. Note that we do not evaluate NaLIR against

Table 7: Overall Accuracy (%)

| Data Set | ATHENA++ | ATHENA | NaLIR |
|---|---|---|---|
| *MAS* | 84.61 | 67.03 | 49.08 |
| *GEO* | 84.25 | 68.20 | 41.04 |
| *Spider* | 78.82 | 54.93 | – |
| *FIBEN* | 88.33 | 48.00 | 20.66 |

the *Spider* data set, since NaLIR requires non-trivial system configurations when switching between relational schemas.

Table 8 presents the accuracy of each NLIDB system specific to the nested queries in each data set. We observe that the winning margin of ATHENA++ over NaLIR and ATHENA is more substantial. In fact, Table 8 further establishes that ATHENA++ is the only NLIDB system among the three that can consistently handle nested queries with a reasonable accuracy. ATHENA++ is able to leverage both linguistic features and domain reasoning, which are very critical for nested query handling as we have outlined in Section 1.

Table 8: Nested Query Accuracy (%)

| Data Set | ATHENA++ | ATHENA | NaLIR |
|---|---|---|---|
| *MAS* | 78.37 | 10.81 | 8.10 |
| *GEO* | 78.57 | 17.14 | 8.57 |
| *Spider* | 78.26 | 9.93 | – |
| *FIBEN* | 85.88 | 15.29 | 7.05 |

Although ATHENA also exploits domain semantics through an ontology, its core interpretation algorithm is not designed to consider nesting. Hence, ATHENA is limited to handling simple nested queries that can be easily translated into non-nested SQL queries. NaLIR heavily relies on linguistic features, which is shown to be insufficient for complex nested queries. Moreover, NaLIR depends on user interactions to resolve ambiguous queries. Its accuracy further drops when the interactive communicator is disabled.

### 6.2.2 Precision, Recall, and F1-score

Next, we study the effectiveness of ATHENA++ nested query detection and generation. We first report the precision, recall, and F1-score of our Nested Query Detector in Table 9. The results show that our nested query detector reliably classifies a NL query as a nested query across all data sets (F1-score $>$ 90% in all cases). We observe that for Spider, which has a much wider variation of linguistic patterns in its nested queries, recall (88.73%) is slightly lower than the other data sets, but precision (95.30%) is still the second best among all four data sets.

Table 9: Effectiveness of Nested Query Detector (%)

| Data Set | Precision | Recall | F1-score |
|---|---|---|---|
| *MAS* | 95.77 | 91.89 | 93.79 |
| *GEO* | 94.11 | 91.42 | 92.75 |
| *Spider* | 95.30 | 88.19 | 91.61 |
| *FIBEN* | 95.23 | 94.11 | 94.67 |

Table 10: Effectiveness of Nested Query Builder (%)

| Data Set | Precision | Recall | F1-score |
|---|---|---|---|
| *MAS* | 81.69 | 85.29 | 83.45 |
| *GEO* | 80.88 | 85.93 | 83.33 |
| *Spider* | 84.56 | 88.73 | 86.59 |
| *FIBEN* | 86.90 | 91.25 | 89.02 |

Next, we report the precision, recall, and F1-score of our Nested Query Builder in Table 10. Note that we only use the nested queries associated with each data set for this evaluation. Clearly, the high precision (83.51% on average) across four data sets further establish that ATHENA++ is able to produce the correct nested queries once it correctly classifies a NL query as a nested query. Moreover, ATHENA++ recognizes a broad range of nested queries leading to a high recall (87.8% on average). Lastly, ATHENA++ demonstrates its consistency across different domains with a wide spectrum of nested queries.

### 6.2.3 Error Analysis

Following the above results, we also provide an error analysis on the queries that are not answered correctly by ATHENA++. Table 11 lists the number of incorrectly answered queries among all the queries in the workloads, with respect to three components where the errors occurred.

Table 11: Error Analysis

| Data Set | Nested Query Detection | Hier. Query Generation | Join Condition Generation |
|---|---|---|---|
| *MAS* | 6 | 8 | 2 |
| *GEO* | 6 | 6 | 3 |
| *SPIDER* | 19 | 10 | 6 |
| *FIBEN* | 10 | 10 | 4 |

Nested query detection and nested query building (i.e., hierarchical query generation and join condition generation) contribute to 47.5% and 52.5% of the errors, respectively. This shows that nested query detection and building are two equally critical challenges in nested query handling. Among the errors in nested query detection and building, we make the following observations.

**Insufficient linguistic patterns.** In some cases, the NL query does not provide sufficient linguistic patterns to be used for detecting a possible nesting. For example, the query *"who invested in the company they work for"* from *FIBEN* does not explicitly mention the company is the *same* between investment and employment. Hence, ATHENA++ fails to detect it as a nested query.

**Heuristic errors.** We also observe that the heuristics introduced in Section 4.3 can cause detection errors. For example, in the query *"which services companies reported lesser revenue in 2019 than the average industry revenue in 2018"* (*FIBEN*), Heuristic 1 fails to detect that *"services companies"* is a shared token with the inner query, as Stanford CoreNLP co-reference resolution does not find any reference from the inner query. On the other hand, in the query *"find the series name and country of the tv channel that is playing some cartoons directed by Ben Jones and Michael Chang"* (*Spider*), Heuristic 4 incorrectly marks the Focus entity (*"series name and country"*) as shared with the inner query. This results in an incorrect join condition.

**Incorrect hierarchical interpretation tree.** As discussed earlier, the heuristics introduced in Section 4.3 do not always choose the correct tokens to share. In some cases, this results in an incorrect hierarchical interpretation tree, even when the join condition is correctly identified. For example, in the query *"what is the distribution by state of the number of people selling Microsoft stock in 2018" (FIBEN)*, the correct join path between *"people"* (referring to *"Person"*) and *"Microsoft"* (an instance of *"Corporation"*) should be via *"FinancialServiceAccount"*, *"Transaction"*, and *"ListedSecurity"* (depicted in Figure 1). However, the Steiner Tree algorithm used in our interpretation tree generator selects a more compact path via *"AutonomousAgent"* using *"isA"* relationships. In this case, the most compact interpretation tree actually leads to an incorrect interpretation.

**Incorrect join conditions.** We find the errors in join condition generation are mainly due to mathematical computations or implicit operations. For example, in the query *"who sold IBM stocks*

*1 month after it had highest buying value in 2016"* (*FIBEN*), Algorithm 3 does not recognize that the tokens *"1 month after"* require a mathematical computation (i.e., month + 1). In the case of implicit operations, the query *"which dogs have not cost their owner more than 1000 for treatment?"* (*Spider*) requires a summation over the cost before comparing with 1000, which is not handled by Algorithm 3.

A few possible approaches to address these issues would be ($i$) exploiting a variety of NLP libraries to extract more and better-quality linguistic patterns, ($ii$) introducing more robust domain reasoning heuristics, and ($iii$) integrating learning-based techniques into ATHENA++ in a multi-step strategy to leverage the best of both worlds (i.e., rule-based and machine learning-based techniques).

### 6.2.4 Performance Evaluation

In this section, we present an analysis of ATHENA++'s SQL generation time. In summary, for the NL queries from 4 workloads, ATHENA++ is able to generate the final SQL query in less than 200 ms, achieving its goal of interactive execution time.

**Table 12: NL-to-SQL Generation Time (ms)**

| Data Set | Type-N | Type-A | Type-J | Type-JA |
|---|---|---|---|---|
| *MAS* | 153 | 158 | 188 | 198 |
| *GEO* | 105 | 118 | 140 | 192 |
| *Spider* | 100 | 91 | – | – |
| *FIBEN* | 108 | 126 | 120 | 144 |

We have excluded the queries for which ATHENA++ is unable to produce the correct interpretation. In Table 12, ATHENA++ generates a SQL query on average within 120 ms for Type-N and Type-A nested query. However, for the more complex Type-J and Type-JA nested queries, ATHENA++ takes 36% more time (163 ms on average) to translate. This is mainly due to the complexity of identifying the correlation between the inner and outer queries.

## 7. RELATED WORK

Natural language interfaces to databases (NLIDBs) has become an increasingly active research field during the last decade. For comprehensive studies of NLIDB systems, we refer to [7, 26, 16]. In general, these systems fall into the following two categories.

**Rule-based systems.** The majority of existing NLIDBs interpret NL queries based on information of the underlying database and then, based on inference rules, the interpretations are mapped into query fragments before finally being assembled into complete queries. NaLIR [20] uses an off-the-shelf parser to obtain a dependency parse tree for a given query. Then, it maps nodes of the parse tree to SQL components. In case of ambiguous NL queries, NaLIR relies on user interaction to find the correct interpretation. PRECISE [28] transforms a given query to Disjunctive Normal Form (DNF), and then consults an inverted index over the contents of a database to look up each disjunct. The result is a logical subset of the original database. ATHENA [29] takes an ontology-driven, two-step approach which separates query interpretation from the actual physical store. This separation helps to reduce the challenges associated with query understanding and translation, as a user may not be familiar with the schema of the underlying database. Recently, TEMPLAR [8] utilizes information from SQL query logs to improve keyword mapping and join path inference. Duoqest [9] leverages guided partial query enumeration to efficiently explore the space of possible queries for a given NL query.

These systems enable users to construct simple database queries such as point queries and basic aggregation queries. In addition, they support limited nesting in natural language queries when the queries explicitly capture different sub-queries, such as *"return all the authors who have more papers than John Doe after 2005"*. However, nested queries are often expressed implicitly in natural language, some originating from linguistic patterns (e.g., *"same as"*, *"different from"*), or logically embedded in comparisons with a sub-query result (e.g., *"more than average"*). To the best of our knowledge, these NLIDBs can only handle nested queries of Type-N and Type-A. In this work, we focus on detecting and handling multiple common nested query types by combining linguistic pattern matching and semantic reasoning over domain schemas.

**Machine learning-based systems.** Recently, a growing number of NLIDBs attempt to leverage advances in deep learning to handle NL queries. The basic idea is to apply supervised machine learning techniques on a set of question/answer pairs where the questions are the NL queries and the answers are the respective SQL statements. These questions and answers are first transformed into a vector by applying word embedding techniques. Then, these vectors are consumed by a deep neural network [39, 34, 10, 35, 12]. Seq2SQL [39] uses deep neural networks to learn generic language patterns in order to translate NL queries to SQL queries. SQLizer [34] presents a hybrid approach that first trains a semantic parser to obtain query sketches and then uses a rule-based approach to repair the sketches into complete SQL queries. DBPal [10, 33] avoids manually labeling large training data sets by synthetically generating a training set that only requires minimal annotations in the database. TypeSQL [36, 35] proposes a different training procedure utilizing types extracted from either knowledge graph or table content to better understand entities and numbers in the query. DialSQL [12] is a dialogue-based structured-query generation framework that leverages human intelligence to boost the performance of existing algorithms via user interaction.

Machine learning-based approaches have shown promising results in terms of robustness to NL variations. The learned models can detect and annotate key SQL components in a given NL query, including select columns, aggregation functions, and where clauses. However, these systems still have limited capability of handling complex queries involving multiple tables with aggregations, and nested queries. In addition, they require large amounts of training data, which makes domain adaption challenging. In this work, we alleviate the above issues by identifying linguistic patterns from NL queries and deeply understanding the domain semantics. As shown in our experiments, ATHENA++ not only handles these complex nested queries with high precision and recall, but also provides a mechanism for domain adaptation without requiring training data.

## 8. CONCLUSION

In this paper, we describe ATHENA++ which translate natural language queries into complex BI analysis queries that often require nesting, join, and aggregation. In particular, we identify the unique challenges for nested query detection and generation w.r.t 4 standard and commonly used nested query types. Our novel system, ATHENA++, combines linguistic patterns from the natural language query with deep domain reasoning using ontologies to enable nested query detection and generation. The experimental evaluation on a variety of benchmark data sets shows that our system outperforms all other state-of-the-art NLIDB systems, and provides consistently high accuracy, precision and recall. In particular, ATHENA++ achieves 78.89% accuracy on the *Spider* dev set, beating the best-reported number (70.6%) by 8%, and achieves 88.33% accuracy on the new *FIBEN* benchmark, which emulates a financial data warehouse with complex analysis queries.

# 9. REFERENCES


[1] FIBO. https://spec.edmcouncil.org/fibo/. Accessed: 2020-03-01.

[2] FRO. http://xbrl.squarespace.com/financial-report-ontology/. Accessed: 2020-03-01.

[3] Microsoft Academic Search. http://academic.research.microsoft.com/. Accessed: 2020-03-01.

[4] Owl 2 web ontology language document overview. https://www.w3.org/TR/owl2-overview/. Accessed: 2020-03-01.

[5] SEC Financial Statement Data Set. https://spec.edmcouncil.org/fibo/. Accessed: 2020-03-01.

[6] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, P. Parag, and S. Sudarshan. Banks: Browsing and keyword searching in relational databases. In *VLDB*, pages 1083–1086, 2002.

[7] K. Affolter, K. Stockinger, and A. Bernstein. A comparative survey of recent natural language interfaces for databases. *VLDB J.*, 28(5):793–819, 2019.

[8] C. Baik, H. V. Jagadish, and Y. Li. Bridging the semantic gap with SQL query logs in natural language interfaces to databases. In *ICDE*, pages 374–385, 2019.

[9] C. Baik, Z. Jin, M. J. Cafarella, and H. V. Jagadish. Constructing expressive relational queries with dual-specification synthesis. In *CIDR*, 2020.

[10] F. Basik, B. Hättasch, A. Ilkhechi, A. Usta, S. Ramaswamy, P. Utama, N. Weir, C. Binnig, and U. Çetintemel. Dbpal: A learned nl-interface for databases. In *SIGMOD*, pages 1765–1768, 2018.

[11] J. Berant et al. Semantic Parsing on Freebase from Question-Answer Pairs. In *EMNLP*, pages 1533–1544, 2013.

[12] I. Gur, S. Yavuz, Y. Su, and X. Yan. Dialsql: Dialogue based structured query generation. In *ACL*, pages 1339–1349, 2018.

[13] J. Hölsch, M. Grossniklaus, and M. H. Scholl. Optimization of nested queries using the nf2 algebra. In *SIGMOD*, page 17651780, 2016.

[14] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. In *ACL*, pages 963–973, 2017.

[15] M. Jammi, J. Sen, A. Mittal, et al. Tooling framework for instantiating natural language querying system. *PVLDB*, 11(12):2014–2017, 2018.

[16] H. Kim, B. So, W. Han, and H. Lee. Natural language to SQL: where are we today? *PVLDB*, 13(10):1737–1750, 2020.

[17] W. Kim. On optimizing an sql-like nested query. *ACM Trans. Database Syst.*, 7(3):443469, Sept. 1982.

[18] D. Küpper, M. Storbel, and D. Rösner. NAUDA: A Cooperative Natural Language Interface to Relational Databases. In *SIGMOD*, pages 529–533, 1993.

[19] C. Lei, F. Özcan, A. Quamar, A. R. Mittal, J. Sen, D. Saha, and K. Sankaranarayanan. Ontology-based natural language query interfaces for data exploration. *IEEE Data Eng. Bull.*, 41(3):52–63, 2018.

[20] F. Li and H. V. Jagadish. Constructing an Interactive Natural Language Interface for Relational Databases. *PVLDB*, 8(1):73–84, 2014.

[21] F. Li and H. V. Jagadish. Understanding natural language queries over relational databases. *SIGMOD Rec.*, 45(1):613, 2016.

[22] Y. Li and D. Rafiei. Natural language data management and interfaces: Recent development and open challenges. In *SIGMOD*, pages 1765–1770, 2017.

[23] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: An Interactive Natural Language Interface for Querying XML. In *SIGMOD*, pages 900–902, 2005.

[24] C. D. Manning, M. Surdeanu, J. Bauer, et al. The Stanford CoreNLP natural language processing toolkit. In *ACL*, pages 55–60, 2014.

[25] M. Nicola, I. Kogan, and B. Schiefer. An xml transaction processing benchmark. In *SIGMOD*, pages 937–948, 2007.

[26] F. Ozcan, A. Quamar, J. Sen, C. Lei, and V. Efthymiou. State of the art and open challenges in natural language interfaces to data. In *SIGMOD*, pages 2629–2636, 2020.

[27] A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *COLING*, pages 141–147, 2004.

[28] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[29] D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. Athena: an ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.

[30] J. Sen, F. Özcan, A. Quamar, G. Stager, A. R. Mittal, M. Jammi, C. Lei, D. Saha, and K. Sankaranarayanan. Natural language querying of complex business intelligence queries. In *SIGMOD*, pages 1997–2000, 2019.

[31] L. R. Tang and R. J. Mooney. Using multiple clause constructors in inductive logic programming for semantic parsing. In *ECML*, pages 466–477, 2001.

[32] S. Tata and G. M. Lohman. Sqak: Doing more with keywords. In *SIGMOD*, pages 889–902, 2008.

[33] N. Weir, P. Utama, A. Galakatos, A. Crotty, A. Ilkhechi, S. Ramaswamy, R. Bhushan, N. Geisler, B. Hättasch, S. Eger, U. Çetintemel, and C. Binnig. Dbpal: A fully pluggable NL2SQL training pipeline. In *SIGMOD*, pages 2347–2361, 2020.

[34] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig. Sqlizer: Query synthesis from natural language. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):63, 2017.

[35] T. Yu, Z. Li, Z. Zhang, R. Zhang, and D. R. Radev. Typesql: Knowledge-based type-aware neural text-to-sql generation. In *NAACL-HLT*, pages 588–594, 2018.

[36] T. Yu, M. Yasunaga, K. Yang, R. Zhang, D. Wang, Z. Li, and D. Radev. SyntaxSQLNet: Syntax tree networks for complex and cross-domain text-to-SQL task. In *EMNLP*, pages 1653–1663, 2018.

[37] T. Yu, R. Zhang, K. Yang, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *EMNLP*, pages 3911–3921, 2018.

[38] J. M. Zelle and R. J. Mooney. Learning to parse database queries using inductive logic programming. In *AAAI*, pages 1050–1055, 1996.

[39] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.