ATHENA: An Ontology-Driven System for Natural Language Querying over Relational Data Stores

Diptikalyan Saha[†], Avrilia Floratou[⋆], Karthik Sankaranarayanan[†], Umar Farooq Minhas[⋆], Ashish R. Mittal[†], Fatma Özcan[⋆]
[†]*IBM Research. Bangalore, India* {diptsaha, kartsank, ashishmittal}@in.ibm.com
[⋆]*IBM Research. Almaden, USA* {avrilia.floratou, ufminhas}@gmail.com, fozcan@us.ibm.com

ABSTRACT

In this paper, we present ATHENA, an ontology-driven system for natural language querying of complex relational databases. Natural language interfaces to databases enable users easy access to data, without the need to learn a complex query language, such as SQL. ATHENA uses domain specific ontologies, which describe the semantic entities, and their relationships in a domain. We propose a unique two-stage approach, where the input natural language query (NLQ) is first translated into an intermediate query language over the ontology, called OQL, and subsequently translated into SQL. Our two-stage approach allows us to decouple the physical layout of the data in the relational store from the semantics of the query, providing physical independence. Moreover, ontologies provide richer semantic information, such as inheritance and membership relations, that are lost in a relational schema. By reasoning over the ontologies, our NLQ engine is able to accurately capture the user intent. We study the effectiveness of our approach using three different workloads on top of geographical (GEO), academic (MAS) and financial (FIN) data. ATHENA achieves 100% precision on the GEO and MAS workloads, and 99% precision on the FIN workload which operates on a complex financial ontology. Moreover, ATHENA attains 87.2%, 88.3%, and 88.9% recall on the GEO, MAS, and FIN workloads, respectively.

1. INTRODUCTION

Natural language interfaces to databases provide a natural way for users to interact with the database. It is gaining further traction with the advent of mobile devices, equipped with strong speech recognition capabilities. A natural language interface is desirable for two reasons: First, it does not require the users to learn a complex query language, such as SQL. Second, the user does not need to know the exact schema of the database; it is sufficient for her to know only the semantic information contained in the database.

There are several challenges in building a natural language interface to databases. The most difficult task is understanding the semantics of the query, hence the user intent. Early systems [7, 34] allowed only a set of keywords, which has very limited expressive power. There have been works to interpret the semantics of a full-blown English language query. These works in general try to disambiguate among the potentially multiple meanings of the words and their relationships. Some of these are machine learning based [5, 29, 33] that require good training sets, which are hard to obtain. Others require user feedback [16, 19, 20]. However, excessive user interaction to resolve ambiguities can be detrimental to user acceptance. Most of the non-learning disambiguation techniques (e.g., [30]) build on database integrity constraints and thus do not capture the rich semantics available in the ontology.

This paper presents ATHENA, an ontology-based system for natural language querying over relational databases. ATHENA uses domain ontologies, which describe the semantic entities and the relationships between them. Ontologies are widely used because not only they capture the semantics of a domain but also provide a standard description of the domain for applications to use.

We propose a unique two-stage approach: In the first stage, a natural language query (NLQ) is translated into an intermediate query language, called Ontology Query Language (OQL), over the domain ontology. In this stage, we propose an interpretation algorithm that leverages the rich semantic information available in the ontology, and produces a ranked list of interpretations for the input NLQ. This is inspired by the search paradigm, and minimizes user's interaction for disambiguation. Using an ontology in the interpretation provides a stronger semantic basis for disambiguation compared to operating on a database schema. It also provides physical independence from the underlying relational database.

The NLQ interpretation engine uses database data and synonyms to map the tokens of the textual query to various ontology elements like concepts, properties, and relations between concepts. Each token can map to multiple ontology elements. We produce an interpretation by selecting one such mapping for each token in the NLQ, resulting in multiple interpretations for a given NLQ. Each interpretation is then translated into an OQL query.

In the second stage, each OQL query is translated into a SQL query by using the mapping between the ontology and database schema. ATHENA does not rely on user interaction to pick the correct interpretation, but rather uses an intuitive ranking function based on ontology metrics to choose the best interpretation. This top-ranked SQL query is run against the database and the results are returned to the user. In addition to the results of the top-ranked query, ATHENA also displays alternative interpretations to the user. The user only needs to choose an alternative interpretation, if the top-ranked query did not capture her intent.

The contributions of the paper can be summarized as follows:

- We present ATHENA, an ontology-based query engine which provides a natural language query interface to relational data. To the best of our knowledge, ATHENA is the first ontology-based NLQ engine for relational databases.
- We propose a unique two-stage approach that first translates an

Proceedings of the VLDB Endowment, Vol. 9, No. 12

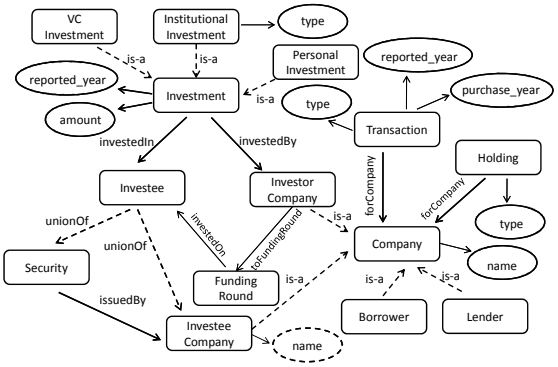

Figure 1: Financial Domain Ontology

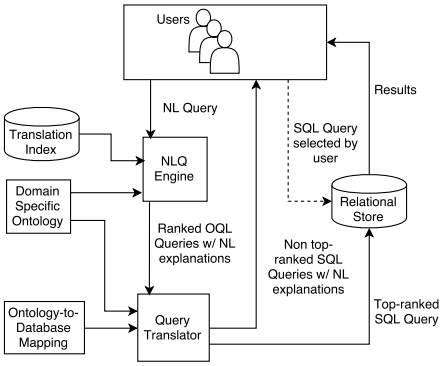

Figure 2: System Architecture

input NLQ into an OQL query defined over the ontology, and then translates the OQL query into its corresponding SQL query.

- We provide a novel ontology-driven algorithm that generates a ranked list of interpretations, and their corresponding OQL queries.
- We describe a novel algorithm to translate the OQL queries into SQL queries. Our algorithm is able to handle various underlying physical representations.
- We show the effectiveness of ATHENA using a comprehensive experimental study on three workloads. ATHENA achieves 100% precision on a geographical (GEO) and an academic (MAS) workload, and 99% precision on a financial (FIN) workload. Moreover, ATHENA attains 87.2%, 88.3%, and 88.9% recall on the GEO, MAS, and FIN workloads, respectively.

2. SYSTEM OVERVIEW

We now present an overview of ATHENA, and show how an example NLQ flows through our system.

2.1 Ontology-Driven Architecture

ATHENA employs a ***Domain Specific Ontology*** (referred to as the ontology) to represent a real-world domain and interprets NLQs using such an ontology. The ontology is expressed in the Web Ontology Language (OWL2) [1].

In this paper, we will use a financial domain ontology, shown in Figure 1, to describe our architecture and algorithms. The ontology contains the *concepts* of the domain (e.g., `Company`) along with their *properties* (e.g., `name`) as well as the *relations* between the concepts (e.g., `forCompany`). Note that Figure 1 shows only a small portion of a complex, real-world ontology that we use internally, and in the experiments we present in Section 4. The full ontology contains 75 concepts, 289 properties and 95 relations. As shown in Figure 1, the ontology contains hierarchies between concepts. For example, an `InvesteeCompany` is a specific type of `Company`, and, thus inherits the properties of the `Company` concept (e.g, `InvesteeCompany.name`). The `InvesteeCompany` is called a *child concept*, the `Company` is called a *parent concept*, and their relationship is captured by an `is-a` arrow (inheritance). Additionally, since Securities and Investee Companies together constitute the collection of Investees, the `Investee` is a *union concept* and `Security` and `InvesteeCompany` are the corresponding *member* concepts, and this relationship is represented by the `unionOf` arrows (membership). Such inheritance and membership relationships are frequently encountered in real-world ontologies.

The data corresponding to the ontology is stored in a ***Relational Store (RS)***. The schema of the RS must capture all the information that is contained in the ontology, and is generated by an RS designer during an offline phase. Typically, multiple relational schemata can conform to the same ontology. For example, the RS designer might create a denormalized, or a normalized schema, or might generate materialized views depending on application requirements. Note that the users of ATHENA are not aware of the relational schema, but form their NLQs relying solely on the ontology. We only require that the RS designer provides our system with an ***Ontology-to-Database Mapping***, which is also generated during an offline phase, and describes how the ontology elements (concepts, properties, and relations) are mapped to database objects (e.g., tables, views, columns, and referential integrity constraints). The mapping must satisfy certain requirements which are discussed in Section 3.1.3. In this paper, we assume that the data corresponding to a particular ontology is already loaded in the RS, and the Ontology-to-Database Mapping is provided by the RS designer.

[1] `https://www.w3.org/TR/owl2-overview/`

2.2 Query Flow

We present the overall system architecture in Figure 2. Assume that the user submits the following NLQ against the ontology presented in Figure 1: *"Show me restricted stock investments in Alibaba since 2012 by investor and year"*. As a first step, the NLQ engine determines which elements of the ontology are referenced by the query. For example, the token "restricted stock" may refer to a value of the property `type` of `InstitutionalInvestment` or `Holding` among others. Similarly, the token "Alibaba" may refer to the `name` of a `Company`, an `InvestorCompany`, or a `Lender`. The NLQ Engine explores all of these options and generates a ranked list of interpretations conforming to the ontological structure and semantic constraints. A natural language explanation is also generated for each such interpretation.

During the query interpretation process, the NLQ Engine relies on an auxiliary service, named the ***Translation Index (TI)***. The TI provides data and metadata indexing for data values stored in the RS, and for concepts, properties, and relations appearing in the ontology, respectively. For our example query above, the NLQ engine would search for the token "Alibaba" in the TI. The TI captures that "Alibaba" is a data value for the `name` column in the `Company` table in the RS, which based on the Ontology-to-Database Mapping, maps to the ontology property `Company.name` (Figure 1). Note that "Alibaba" maps to multiple ontology elements (e.g., `InvestorCompany` or `Lender`), and the TI captures all of them. TI provides powerful and flexible matching by using semantic variant generation schemes. Essentially, for the data values indexed in the TI, we not only index the actual values (e.g., distinct values appearing in `Company.name`), but also variants of those distinct values. We support semantic variant generators (VGs) for person and company names, among others. For example, given an input string "Alibaba Inc", the company name VG produces the fol-

lowing list of variants: {*"Alibaba", "Alibaba Inc", "Alibaba Inc.", "Alibaba Incorporated"*}. This allows the users of ATHENA to formulate the queries by using any of the indexed variants of a data value (e.g., "Alibaba" vs. "Alibaba Inc"). The TI is built during an offline initialization phase, and is populated from the RS.

One key distinguishing feature of ATHENA is the use of a two-stage approach that provides *physical independence*. By solely reasoning over the ontology, and by exploiting the TI, our NLQ engine is completely oblivious to the actual representation of the data in the RS. To support this two-stage approach, we define an intermediate query language over the ontology, namely ***Ontology Query Language (OQL)***. The role of OQL is to provide independence from the underlying data stores and their target languages. For example, the same query interpretation can be executed against a relational store and/or a graph store. This paper focuses on Relational Stores, and thus OQL serves as an input to the ***Query Translator*** which generates the corresponding SQL queries. Each interpretation generated by the NLQ engine is translated into an OQL query.

The Query Translator takes the ranked OQL queries, along with the ontology and the corresponding Ontology-to-Database Mapping, and generates a list of SQL queries. As opposed to previous systems (e.g., [19]), ATHENA automatically picks the top-ranked SQL query, submits it to the RS for execution and presents the results to the user. As shown in Figure 2, the complete list of ranked SQL queries with their natural language explanations [22] are also presented to the user. Thus, the user can choose an alternative query interpretation (SQL) and execute it against the RS, if the top-ranked query did not capture her intent. As we show in Section 4, for the majority of the queries we tested, the top-ranked query interpretation is actually the intended interpretation, and only in some rare cases does the user need to pick an alternative interpretation.

3. ALGORITHMS

We now present the algorithms used by ATHENA. We start by describing the notations used in these algorithms.

3.1 Notations

3.1.1 Ontology

An Ontology $O = (C, R, P)$ contains a set of concepts $C = \{c_n, 1 \leq n \leq N\}$, a set of relations $R = \{r_k, 1 \leq k \leq K\}$ and a set of properties $P = \{p_m, 1 \leq m \leq M\}$ that represent a real-world domain. The ontology domain consists of real-world entities called *individuals* which are grouped into concepts based on similarity of characteristics. A *property* represents a characteristic of a concept and belongs uniquely to that concept. P_n is the set of properties belonging to concept $c_n \in C$ and P is the set of all properties. We use the naming convention $c.p$ to refer to the property p that belongs to concept c ($c.p \in P$). Each relation $r_k = (c_i, c_j) \in R$ represents a relationship between the concepts c_i and $c_j \in C$. We use the term *ontology element* to refer to a concept, property or relation of the ontology. Each *ontology element* is associated with a set of *synonyms* which are generated manually or semi-automatically [12, 21].

The relations have an associated type, namely *membership*, *inheritance*, and *functional*. The set R_M contains the *membership* relations where $(c_i, c_j) \in R_M$ denotes that the concept c_i is a *union* concept and the concept c_j is a *member* concept. The set R_I contains the *inheritance* relations where $(c_i, c_j) \in R_I$ denotes that the concept c_i is a *parent concept* and that the concept c_j is a *child concept*. Note that unlike a *union concept*, a *parent concept* can contain individuals that are not present in any of its child concepts. However, as noted in Section 2.2, the child and the member concepts inherit the properties of their parent or union concepts, respectively. Thus, the set P includes the inherited properties of all the child and member concepts. Finally, the set R_F contains the remaining relations of the ontology which are called *functional* relations. The set of the ontology relations R can be defined as: $R = R_M \cup R_I \cup R_F$.

3.1.2 Ontology Query Language (OQL)

The OQL language[2] is specifically designed to express queries over an ontology and is agnostic to the underlying physical schema. The focus of this paper is to explain how our system achieves physical independence through the OQL language and not to present OQL's functionality in detail. The portion of the OQL grammar that we use in this paper is presented below:

```
UnionQuery:    Query (UNION Query)*
Query:         select from where?  groupBy?  orderBy?  having?
select:        (aggrType?(PropertyRef))+
from:          (Concept ConceptAlias)+
where:         binExpr1* binExpr2* inExpr?
groupBy:       (PropertyRef)+
orderBy:       (aggrType?(PropertyRef))+
having:        aggrType(PropertyRef) binOp value
value:         Literal+ | Query
aggrType:      SUM| COUNT| AVG | MIN | MAX
binExpr1:      PropertyRef binOp [any] value
binExpr2:      ConceptAlias RelationRef+ = ConceptAlias
inExpr:        PropertyRef IN Query
binOp:         > | < | >= | <= | =
PropertyRef:   ConceptAlias.Property
RelationRef:   Relation ->
```

The OQL grammar can express sophisticated OLAP-style queries that include aggregations, unions, and nested subqueries. An OQL query operates on top of individuals of concepts where each concept has an alias defined in the `from` clause of the query. Section 3.2.3 presents more details about OQL.

3.1.3 Ontology-to-Database Mapping

As noted earlier, our system architecture is independent of the underlying database schema which allows an RS designer to incorporate any existing database without explicitly modeling the ontology elements. The RS designer creates the Ontology-to-Database Mapping which is given as input to ATHENA, and describes how the ontology elements are mapped to the database objects.

A relational database $D = (T, F, S)$ consists of a set of tables T, a set of fields F, and a set of referential integrity constraints S. The term *table* refers to both database tables and views. A field belongs uniquely to a table or a view, and F is the set of all fields. The set S contains the referential integrity constraints between tables in T.

The *Ontology-to-Database Mapping* $ODM(O, D) = (\overrightarrow{T}, \overrightarrow{F}, \overrightarrow{S})$, maps an ontology $O = (C, R, P)$ to a database $D = (T, F, S)$ through the mapping functions $\overrightarrow{T}$, $\overrightarrow{F}$, and $\overrightarrow{S}$. The mapping functions are provided to our system by the RS designer and they must satisfy certain requirements that we describe next. Firstly, each concept in the ontology must be mapped to one database table or view through the function $\overrightarrow{T} : C \mapsto T$. ATHENA maps each concept's synonyms to the same database table or view. The function $\overrightarrow{F} : P \mapsto F$ maps the ontology properties to their corresponding database fields. Since the ontology allows for *inheritance* and *membership* relations, our system offers two alternatives. In the first alternative, the database designer can map the properties of the child, member or union concepts to fields in the database directly through the function $\overrightarrow{F}$. Alternatively, ATHENA will derive them indirectly from the corresponding properties of the parent, union, or member concepts. A table or view that represents a concept that is not a child, a member

[2]Note that the language is different from the Object Query Language: https://en.wikipedia.org/wiki/Object_Query_Language

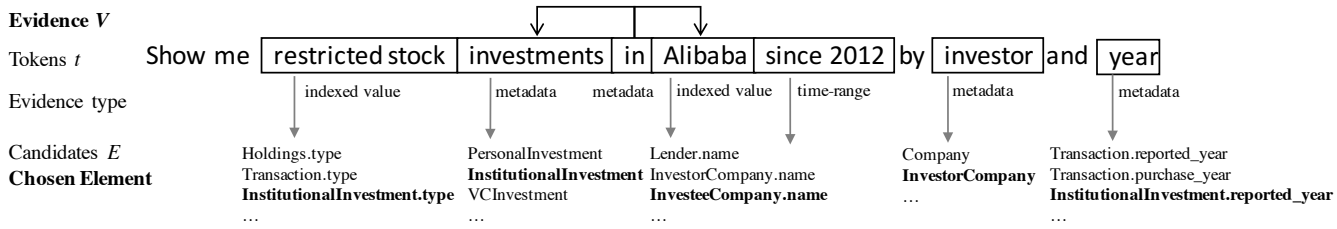

Figure 3: A Natural Language Query annotated with the Evidence Set V obtained from the Ontology Evidence Annotator.

or a union concept, must include all the database fields that correspond to this concept's properties. Note that this model does not pose any restriction on the database schemata supported. This is because the RS designer can map concepts to arbitrarily complex view definitions which incorporate database fields that might belong to different tables. Finally, each relation between concepts of the ontology must be mapped to a set of referential integrity constraints between database tables through the function $\overrightarrow{S} : R \mapsto 2^S$.

3.2 Natural Language Query (NLQ) Engine

The NLQ Engine consists of 3 components discussed next.

3.2.1 Ontology Evidence Annotator

As a first step, the NLQ engine invokes the *Ontology Evidence Annotator* component on the input NLQ text. Each annotation produced by this component, shows evidence that one or more ontology elements (concept, relation, property) have been referenced in the input NLQ. Formally, an *evidence* $v : t \mapsto E$ maps a token t (set of words in the NLQ text) to a set of ontology elements $E \subseteq \{C \cup R \cup P\}$ called *candidates*. Figure 3 shows the candidates obtained for our example NLQ. The Ontology Evidence Annotator consists of the following two major types of annotators:

- **Metadata Evidence Annotator**: This annotator performs a lookup against a *dictionary* which maps the synonyms of the ontology elements (see Section 3.1.1) to the ontology elements. This process produces a set of *metadata evidences* where tokens in the NLQ are associated with ontology elements whose synonyms match those tokens. For example, in Figure 3, the token "investments" maps to the `InstitutionalInvestment` and `PersonalInvestment` concepts among others. As a rule, an annotation that spans a set of contiguous tokens W in the text is preferred over an annotation that spans a subset of W.
- **Data Value Evidence Annotator**: The Data Value Annotators associate data values referenced in the NLQ with a set of ontology properties, thus producing *data value evidences*. Our system supports three kinds of annotators. The *Indexed Value Annotator* annotates data values that are indexed by the Translation Index (TI) (*indexed value evidence*). For example, the token "Alibaba" in Figure 3 can be matched to various properties including the `Lender.name` property, according to the TI. The *Time Range Expression Annotator* annotates tokens in the NLQ that indicate time ranges (e.g, "since 2012") using the TIMEX annotator [35] (*time range evidence*). These tokens are then associated with the ontology properties whose corresponding data type is time-related (e.g., Date). Finally, the *Numeric Expression Annotator* annotates tokens that mention numeric quantities, either in the form of numbers ("45.3") or in text ("fifty two") using the Stanford Numeric Expressions annotator [23] (*numerical evidence*). The tokens are subsequently matched to ontology properties with numerical data types (e.g., Double).

The Ontology Evidence Annotator is also responsible for annotating dependencies between tokens in the NLQ. For example, the NLQ shown in Figure 3 contains a dependency between the tokens "investments" and "Alibaba" through the token "in". These dependencies are called *Relationship Constraints* and are formally defined in the following section.

The output of the Ontology Evidence Annotator is a set of *Relationship Constraints RC* and an *Evidence Set* V. Each evidence $v_i : t_i \mapsto E_i \in V$ has a type (*metadata* or *data value* evidence) and maps a token t_i to a set of candidates E_i. Some tokens other than prepositions, adverbs, conjunctions or tokens in the head phrase (e.g., "show me", "tell me", etc.) may fail in mapping to any ontology element. In this case, our system stops the interpretation process and returns an appropriate message to the user.

3.2.2 Ontology-driven Interpretations Generator

At the next step, ATHENA invokes the *Ontology-driven Interpretations Generator* component. This component uses a novel ontology-driven interpretation algorithm to generate a ranked list of interpretations for the input NLQ. Our algorithm first computes a set of *selected sets* from the *Evidence Set* V. A *selected set* (*SS*) is formally defined as $SS = \{(t_i \mapsto e_i) \mid \forall (t_i \mapsto E_i) \in V, \exists\, e_i \in E_i\}$. Informally, it is formed by iterating over all evidences $(t_i \mapsto E_i)$ in V and collecting a **single** ontology element (e_i) (called the *chosen element*) from each evidence's candidates (E_i). Figure 3 shows the chosen elements of one possible *selected set* (in bold) for our example NLQ. Our algorithm attempts to find one interpretation for each *selected set*. Then, it computes the interpretations corresponding to all the *selected set*s and subsequently ranks them. Each interpretation is represented by *a set of interpretation trees*. Figure 4 shows an interpretation consisting of two *interpretation trees* associated with the *selected set* of Figure 3. These trees will be further discussed towards the end of Section 3.2.2.2. The scenarios where an interpretation contains multiple *interpretation trees* are discussed later. We now formally define an *interpretation tree*.

3.2.2.1 Interpretation Trees.

An *interpretation tree* (*ITree*) corresponding to an Ontology $O = (C, R, P)$ and a *selected set* (*SS*) is defined as $ITree = (C', R', P')$ such that $C' \subseteq C$, $R' \subseteq R$, and $P' \subseteq P$. The *ITree* must satisfy the following constraints:

- **Evidence Cover**: The *ITree* must contain the *chosen element* of each evidence in *SS*. This constraint ensures that the *ITree* covers all the annotated tokens of the NLQ.
- **Weak Connectedness**: The *ITree* must satisfy the weak connectedness property of directed graphs [1]. This property states that the *undirected* graph created by removing the direction of the relation edges in the *ITree* must be connected. More specifically, all the concepts in the *ITree* must be connected to each other through an undirected path of relation edges, and each property must be connected to its corresponding concept. This constraint avoids forming cartesian products when generating the

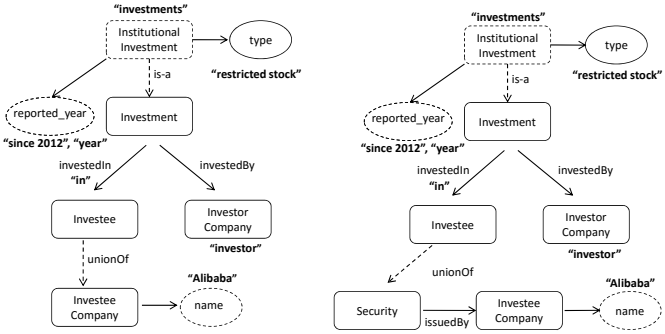

Figure 4: Interpretation trees that form the top-ranked interpretation for the NLQ of Figure 3.

SQL query that corresponds to the *ITree*. If only directed paths are considered, we may miss valid interpretations. For example, consider that concepts A, B, C are connected with the path $A \leftarrow B \rightarrow C$ in the ontology. A valid join path exists between A and C's corresponding database tables even though there is no directed path connecting concepts A and C.

- **Inheritance Constraint**: Consider a modified tree of Figure 4(a). In this tree, the token "restricted stock" is still mapped to the `InstitutionalInvestment.type` property but the token "investments" is now mapped to the `Investment` concept. Note that in our ontology, the `InstitutionalInvestment` concept is a child of the `Investment` concept, and the `type` property is specific to the `InstitutionalInvestment` concept, and not its parent concept. This tree implies that investments have a type ("restricted stock" type), essentially requiring the `Investment` concept to inherit the `type` property from its child concept. The Inheritance Constraint invalidates such trees by prohibiting a *chosen element* who is a parent (or union) concept to inherit a property or a relation from a *chosen element* that corresponds to its child (or member) concept.
- **Relationship Constraint**: A Relationship Constraint is a triple of *tokens* $\langle t_1, t_2, t_3 \rangle$ whose corresponding *chosen elements* e_1, e_2, e_3 in SS satisfy the following constraints: $e_1, e_3 \in C \cup P$ and $e_2 \in R$. The *ITree* satisfies the Relationship Constraint if the relation e_2 is contained in the undirected path between the nodes e_1 and e_3. As noted in the *ITree* shown in Figure 4(a), the *chosen elements* corresponding to the "investments", "in" and "Alibaba" tokens are the `InstitutionalInvestment`, `investedIn` and `InvesteeCompany.name` ontology elements respectively. This *ITree* satisfies the ("investments", "in", "Alibaba") constraint since the undirected path between the `InstitutionalInvestment` and `InvesteeCompany.name` ontology elements contains the `investedIn` relation. Let us consider a slightly different tree, in which the token "Alibaba" is mapped to the `InvestorCompany.name` property. This tree does not satisfy the Relationship Constraint since the path between the `InstitutionalInvestment` and `InvestorCompany.name` ontology elements does not contain the `investedIn` relation.

3.2.2.2 Ontology-driven Interpretation Algorithm.

We now present our interpretation algorithm which takes as input the *Evidence Set V* and a set of *Relationship Constraints RC* and generates a ranked list of interpretations for the input NLQ. Our algorithm builds on techniques previously used in keyword search systems such as [2, 7, 14, 34]. However, it significantly extends these techniques to better exploit the richer semantics present in the ontology in the form of inheritance and membership relations. In summary, our algorithm first creates all *selected sets* from the *Evidence Set V*. For each *selected set*, the algorithm attempts to generate a **single** interpretation that consists of **one or more** *ITrees* that satisfy the constraints presented before. Finally, the interpretations corresponding to different *selected sets* are ranked.

Our algorithm first creates an *undirected graph G* by removing the direction of the relation edges in the ontology graph. Moreover, multiple edges between two ontology concepts are replaced by a single edge in G. The algorithm operates on the undirected graph because this enables finding more solutions that satisfy via the Weak Connectedness Constraint. Corresponding to each *selected set*, the algorithm attempts to identify an *undirected interpretation tree* (*UITree*) which is a *connected* tree of the graph G that satisfies the Evidence Cover, Inheritance, and Relationship constraints. For example, a valid *UITree* is produced if we ignore the direction of the edges of the tree shown in Figure 4(a). As we show next, the *UITrees* will later be converted to directed *ITrees*.

A *UITree* must contain all the *chosen elements* in the *selected set* (Evidence Cover). Note that there can be many connected trees of G spanning all the *chosen elements*. Our system will choose the tree that has the minimal number of functional relations. As noted in [7, 34], the problem is similar to the Steiner Tree problem. Given an undirected weighted graph, and a set of nodes called Steiner Nodes, a Steiner Tree is a tree with *minimal* total weighting for its edges that spans all the Steiner Nodes through additional intermediate nodes, and edges if needed [28]. Note that the Steiner Tree problem is NP-Complete and is typically addressed by approximation algorithms (e.g., [15]). In our case, the Steiner Nodes consist of the concepts, properties, and source and destination concepts of the relations in the current set of *chosen elements*. The typical Steiner Tree algorithms can produce a connected tree that satisfies the Evidence Cover constraint. However, these algorithms do not guarantee that the resulting tree will satisfy the Inheritance and Relationship constraints. To handle these constraints, we first preprocess the graph G to make sure the Inheritance Constraint will be satisfied, and then we invoke a modified Steiner Tree algorithm which takes the Relationship Constraints into account.

Given a *selected set*, our interpretation algorithm creates the set of Steiner Nodes as described above. It, then, preprocesses the graph G in two phases to generate the graph G'. In the first phase, the algorithm assigns appropriate weights to the graph edges (note that the Steiner Tree algorithms operate on weighted graphs). More specifically, the edges that correspond to relations which are *chosen elements* have a weight equal to 0. In this way, we can make sure that these edges will be included in the resulting *UITree*. The remaining edges that correspond to functional relations have a weight equal to 1. The edges that correspond to inheritance and membership relations have a weight equal to 0. This ensures that the inherited functional relations of child and member concepts have the same weight as their non-inherited functional relations.

In the second phase, the algorithm removes certain edges from the graph to make sure that the Inheritance Constraint is satisfied. More specifically, all the edges of G that represent inheritance and membership relations are removed, if the corresponding parent or union concept is a *chosen element*. Let's take the example tree of Section 3.2.2.1, where the token "investments" is mapped to the `Investment` concept but the token "restricted stock" is mapped to the `InstitutionalInvestment.type` property. This tree violates the Inheritance Constraint. Since in this case the parent concept `Investment` is a *chosen element*, our algorithm removes the edge from G that connects the `Investment` and `InstitutionalInvestment` concepts. As a result, the invalid

Algorithm 1: Modified Steiner Tree Algorithm

Input: Graph G', Steiner Nodes *SN*, Relationship Constraints *RC*
Output: *UITree* or Null

1 Set of paths P={}
2 **foreach** *Relationship Constraint* (t_1,t_2,t_3) *in RC with corresponding chosen elements* e_1,e_2,e_3 **do**
3 Choose a *relationship path* $p(e_1,e_3)$ that connects the nodes e_1 and e_3 through the relation e_2 in the graph G' and add $p(e_1,e_3)$ to P
4 **if** *the graph that contains all the paths in P has a cycle* **then**
5 backtrack
6 **end**
7 **end**
8 **if** *a set P is not found* **then**
9 return Null
10 **end**
11 Create a complete undirected graph G_S whose vertices are the Steiner Nodes *SN*.
12 For every $p(e_i,e_j) \in P$, assign negative weight to the edge between nodes e_i and e_j in G_S.
13 For the remaining elements $e_i,e_j \in SN$ that are not part of a Relationship Constraint, set the weight of the edge between e_i and e_j in G_S equal to the weight of the shortest path from e_i to e_j in G'.
14 Find the minimal spanning tree T of G_S
15 Form G_T by replacing T's edges by the corresponding shortest paths and relationship paths, giving a weight of -1 to the edges of the relationship paths.
16 *UITree* = Invoke the cycle removal, unnecessary node and edge removal steps of the algorithm in [15] on G_T.
17 return *UITree*

connected tree will not be generated.

After generating the graph G', our interpretation algorithm invokes an approximation algorithm which extracts a connected tree from G' that spans all the Steiner Nodes and satisfies the Relationship Constraints. Our algorithm is a modification of a well-known Steiner Tree approximation algorithm [15] which additionally guarantees that the resulting tree satisfies the Relationship Constraints.

The algorithm in [15] creates a complete undirected graph G_S that contains only the Steiner Nodes. Each edge between two Steiner Nodes in G_S corresponds to the shortest path between them in the graph G'. The weight of the edge is equal to the sum of the weights of the edges in the shortest path. Next, a minimal spanning tree T of G_S is computed. Then, T's edges are replaced by the corresponding shortest paths to form G_T. The algorithm removes possible cycles, unnecessary nodes, and edges from G_T, and produces a tree that approximately has the minimal weight.

Our modified Steiner Tree algorithm (presented in Algorithm 1) first modifies the formation of graph G_S. For the *chosen elements* that are part of a Relationship Constraint, a path that satisfies the Relationship Constraint is chosen (as short as possible) instead of the shortest path. We call this path *relationship path*. If there are multiple Relationship Constraints, our algorithm attempts to select the *relationship paths* such that the graph that contains all the paths does not contain cycles (Lines 1-10). The weight of the corresponding edges in G_S is negative so that these edges are included in T (Line 12). G_T is formed by replacing T's edges with the corresponding shortest paths, and the *relationship paths*. Note that the edges of the *relationship paths* have a weight equal to -1 in G_T which guarantees that they are included in the final tree (Line 15).

After a *UITree* has been generated for a given *selected set*, our interpretation algorithm examines whether the tree contains membership relations. In case there are multiple paths in G' between a union concept and another concept of the *UITree* through the union concept's member nodes then multiple *UITrees* can be generated. As shown in the *selected set* of Figure 4(a) the token "Alibaba" is mapped to the `InvesteeCompany` concept. According to Figure 1, the `InvesteeCompany` concept is connected to the `Investee` union concept either directly or through the `Security` concept. The tree of Figure 4(a) connects the two concepts directly. However, as depicted in the ontology, another way of investing in "Alibaba" is by purchasing securities issued by the company. Thus, our algorithm generates the tree of Figure 4(b) which contains the `Security` concept (the details are ommitted in the interest of space). These two trees form the interpretation that corresponds to the *selected set*.

The last task is to convert the *UITree*s corresponding to the *selected set* into *ITrees* by mapping their undirected edges to the directed edges of the ontology. As noted before, an undirected edge in the *UITree* can be mapped to multiple directed edges of the ontology graph. Unless there exists one directed edge that corresponds to a relation which is a *chosen element*, all the directed edges are considered. Therefore, one *UITree* can be mapped to more than one *ITrees* which all become part of the interpretation.

After an interpretation for each *selected set* is generated, the interpretations are ranked based on the number of distinct functional relations contained in each interpretation's constituent *ITrees*.

3.2.3 Ontology Query Builder

As described in the previous section, our interpretation algorithm produces a ranked list of interpretations for a given NLQ. The goal of the *Ontology Query Builder* is to represent each interpretation in that list as an OQL query. In this section, we provide a high-level overview on the OQL query generation process.

As discussed in the previous section, each interpretation consists of a set of *interpretation trees* and a *selected set*. The OQL query that corresponds to an interpretation is a union of individual OQL queries, each one associated with one *interpretation tree*. Our example NLQ will produce an OQL query that is a union of two OQL queries, each one associated with one of the interpretation trees shown in Figure 4. The OQL query related to the first interpretation tree is shown in Figure 5(a). The constituent clauses of an OQL query corresponding to a given *interpretation tree* (*ITree*) and a *selected set* (*SS*) are generated as follows:

- **`from` clause:** The `from` clause specifies the concepts referenced in the OQL query along with their aliases. Our algorithm iterates over all the evidences in *SS* and creates an alias for each concept associated with the *chosen element* of each evidence. As shown in Figure 3, the *chosen elements* are associated with the `InstitutionalInvestment`, `InvesteeCompany` and `InvestorCompany` concepts. Thus, the OQL query of Figure 5(a) contains 3 concept aliases to represent these concepts.
- **`groupBy` clause:** The `groupBy` clause specifies an ordered list of ontology properties that the user wishes to group the results by (grouping properties). We treat the presence of the word *by* in the NLQ as a trigger word, possibly indicating that the user wishes to group by the terms following it. Consider a slight variant of the NLQ of Figure 3: *"Show me investments by Alibaba by investor and year"*. In this case, the user wants to group the results first by "investor" and then by "year". Thus, the token "Alibaba" that follows the word *by* should not be considered for grouping.

  Our algorithm is able to make such token differentiations based on the type of each evidence in the *selected set SS*. In our example, the token "Alibaba" produces a *data value* evidence whereas the other two tokens produce a *metadata evidence*. Our algorithm considers only the *metadata evidences* whose *chosen element* is a concept or a property as potential candidates for grouping. Concepts are represented in the `groupBy` clause by their *key* property which is configured to denote the concept's unique identifier (similar to the primary key of a relational table). For example, the `name` property is the *key* property of the `InvestorCompany` concept. Our system then applies lexical and dependency rules on the token of each evidence to determine whether the corresponding *chosen element*

```
SELECT    Sum(oInstitutionalInvestment.amount),
          oInvestorCompany.name,
          oInstitutionalInvestment.reported_year
FROM      InstitutionalInvestment oInstitutionalInvestment,
          InvestorCompany oInvestorCompany,
          InvesteeCompany oInvesteeCompany
WHERE     oInstitutionalInvestment.type = 'restricted stock'
          oInstitutionalInvestment.reported_year >= '2012',
          oInstitutionalInvestment.reported_year <= Inf,
          oInvesteeCompany.name = {'Alibaba Holdings Ltd.',
               'Alibaba Inc.', 'Alibaba Capital Partners'},
          oInstitutionalInvestment→isa→InvestedBy = oInvestorCompany,
          oInstitutionalInvestment→isa→investedIn→unionOf_Security
                                   →issuedBy = oInvesteeCompany
GROUP BY  oInvestorCompany.name,
          oInstitutionalInvestment.reported_year
```
(a)

```
SELECT    Sum(rInstitutionalInvestment.amount),
          rInvestorCompany.name,
          rInstitutionalInvestment.reported_year
FROM      RInstitutionalInvestment rInstitutionalInvestment,
          RInvestorCompany rInvestorCompany,
          RInvesteeCompany rInvesteeCompany,
          RInvestment rInvestment, RSecurity rSecurity
WHERE     rInstitutionalInvestment.type = 'restricted stock' and
          rInstitutionalInvestment.reported_year >= '2012'and
          rInvestorCompany.name = {'Alibaba Holdings Ltd.',
               'Alibaba Inc.', 'Alibaba Capital Partners'} and
          rInstitutionalInvestment.id = rInvestment.id and
          rInvestment.investedBy = rInvestorCompany.id and
          rInvestment.investedIn = rInvestee.id and
          rInvestee.securityId = rSecurity.id and
          rSecurity.issuedBy = rInvesteeCompany.id
GROUP BY  rInvestorCompany.name,
          rInstitutionalInvestment.reported_year
```
(b)

```
SELECT    Sum(rInvestment.amount),
          rCompany2.name, rInvestment.reported_year
FROM      RInstitutionalInvestment rInstitutionalInvestment,
          RInvestorCompany rInvestorCompany,
          RInvesteeCompany rInvesteeCompany,
          RInvestment rInvestment, RCompany rCompany1,
          RCompany rCompany2, RSecurity rSecurity
WHERE     rInstitutionalInvestment.type = 'restricted stock' and
          rInvestment.reported_year >= '2012'and
          rCompany1.name = {'Alibaba Holdings Ltd.',
               'Alibaba Inc.', 'Alibaba Capital Partners'} and
          rInstitutionalInvestment.id = rInvestment.id and
          rInvestment.investedBy = rInvestorCompany.id and
          rInvestment.investedIn = rInvestee.id and
          rInvestee.securityId = rSecurity.id and
          rSecurity.issuedBy = rInvesteeCompany.id and
          rInvesteeCompany.id = rCompany1.id and
          rInvestorCompany.id = rCompany2.id
GROUP BY  rCompany2.name, rInvestment.reported_year
```
(c)

Figure 5: An OQL sub-query (a), and the corresponding SQL queries (b) and (c) on different relational schemata

is a grouping property. According to the lexical rule, if the token immediately follows the trigger word, then the *chosen element* is included in the `groupBy` clause. Additionally, the Stanford Dependency Parser [11] is employed to capture long range dependencies (spanning two or more words) between a token and the trigger word. In case a dependency exists, the corresponding *chosen element* is also a grouping property. According to these rules only the `InvestorCompany.name` and `InstitutionalInvestment.reported_year` properties qualify as grouping properties for the NLQ shown in Figure 3.

- **`select` clause:** The `select` clause contains a list of ontology properties which are categorized as *aggregation properties* and *display* properties depending on whether an aggregation function is applied to them. In Figure 5(a) the property `oInvestorCompany.name` is a display property whereas the property `oInstitutionalInvestment.amount` is an aggregation property. The first challenge when forming the `select` clause is to determine whether it will include aggregation properties or not. Our system identifies *explicit* references of aggregation functions (e.g., SUM) in the NLQ by employing a lexicon of terms corresponding to the aggregation functions supported (see Section 3.1.2). However, in many cases there are *implicit* references of an aggregation function. For example, the existence of `groupBy` properties in the NLQ of Figure 3, implies that the `select` clause should contain an aggregation. In this example, the aggregation is related to the "investments" token and as a result the `InstitutionalInvestment` concept. In such cases, we use a default aggregation property and aggregation function for the concept involved. As shown in Figure 5(a), the default aggregation function is SUM and is applied on the `amount` property.

  In case the query does not include any aggregation, the algorithm iterates over the *metadata evidences* and creates one display property for each evidence whose *chosen element* is a property or a concept. Note that similar to SQL, the `groupBy` properties are included in the `select` clause as display properties. The queries that include aggregation are more challenging since depending on the ontology, we may encounter the "double counting" problem. This problem is well-known in the context of OLAP [18, 27] and we will not discuss it extensively. Consider two concepts c_1 and c_2 in the *ITree*, connected through a many-to-one relation r. If we aggregate over a property of concept c_2 iterating through its individuals by following the path r starting from concept c_1, we will unavoidably count the same individual of c_2 multiple times, thus computing an incorrect aggregation. Our system is able to identify when an aggregation is "unsafe" and generates a nested query (see `inExpr` in Section 3.1.2) to avoid the "double counting" problem. In case the aggregation is "safe", the aggregation property is added to the `select` clause.

- **`orderBy` clause:** The `orderBy` clause specifies an ordered list of ontology properties (or aggregations over properties) that the user wishes to order the results by. We use trigger words (e.g., *least*, *most*, *ordered by*, *top*) to identify whether the user expects the results to be ordered. Similar to the `groupBy` clause, we use only the *metadata evidences* whose *chosen element* is a concept or a property as potential candidates for ordering. We represent the concepts in the `orderBy` clause by their *key* property.
- **`where` clause:** As noted in Section 3.1.2, the `where` clause consists of `binExpr1` and `binExpr2` expressions. The `binExpr1` expressions specify predicates applied on a property, and are generated based on the *data value evidences* in *SS*. For example, the predicate on the `oInvesteeCompany.name` property shown in Figure 5(a) is a `binExpr1` expression. Each `binExpr1` expression consists of the property that corresponds to the particular evidence, an operator, and a set of literal values. For *indexed value evidences*, the operator is the equality ('=') operator and the literal values are those obtained from the TI. For example, the "Alibaba" variants of our example OQL query are obtained from the TI. As shown in Figure 5(a), the *time range evidences* produce two `binExpr1` expressions referring to the start and end timestamp. For *numerical evidences*, the operator is set based on the comparator phrase immediately preceding the numeric expression in the NLQ. If the operator is applied on an aggregation over the property under consideration, instead of generating a `binExpr1` expression, we generate a **`having` clause** that specifies a predicate on the aggregation quantity.

  The `binExpr2` expressions denote how the concepts in the `from` clause are connected to each other in the *ITree*. Each `binExpr2` expression specifies a list of relations that connect the concept alias on the left-hand side *(lConcept)* to the concept alias on the right-hand side *(rConcept)*. We create `binExpr2` expressions that connect the concept alias that corresponds to the root node of the *ITree* (as *lConcept*) to every other concept alias in the `from` clause (as *rConcept*). In Figure 5(a), the concept alias `oInstitutionalInvestment` corresponding to the root node is connected to the `oInvestorCompany` and `oInvesteeCompany` concept aliases through two `binExpr2` expressions.

3.3 Query Translator

In this section, we describe the Query Translator which is responsible for converting an OQL query over an ontology O into a SQL query over a database D, using the *Ontology-to-Database Mapping* $ODM(O,D) = (\overrightarrow{T}, \overrightarrow{F}, \overrightarrow{S})$. For the rest of this section, we use both a *normalized* and *denormalized* relational schema for the ontology shown in Figure 1. To keep our example simple, both schemata contain one table for each concept, named after the concept with the prefix "R" (e.g., the `RCompany` table repre-

sents the Company concept). However, as noted in Section 3.1.3, ATHENA can support much more complex mappings. Each table contains a column for each property of the concept that it represents, and a primary key column, named id. In the *normalized* schema, the tables that represent child concepts contain only the id column and are connected to the tables of their corresponding parent concept through a foreign key (FK) constraint. Thus, to retrieve the values of the InvestorCompany.name property, we join the RInvestorCompany and RCompany tables and then perform a selection on the RCompany.name column. In the *denormalized* schema, the tables that represent child concepts contain additional columns for all the properties that are inherited from their parent concept. Thus, a selection on the RInvestorCompany.name column returns the values of the InvestorCompany.name property. Figures 5(b) and 5(c) show the SQL queries produced by the Query Translator for the OQL query of Figure 5(a) on the *denormalized* and *normalized* schema respectively.

3.3.1 Representation of OQL Queries

Given an OQL query that corresponds to a single NLQ interpretation, the Query Translator identifies the ontology elements referenced in the OQL query as well as the operations applied to them.

The Query Translator represents an OQL query as a set of *attributes* and a set of *join conditions*. An *attribute* represents a particular PropertyRef of the OQL query (see Section 3.1.2), such as oInvestorCompany.name in the query of Figure 5(a). Formally, the *attribute* a = (c, *alias*, p) references the concept c (e.g., InvestorCompany) through the alias *alias* (e.g., oInvestorCompany), and its property p (e.g., name). The concept c is the attribute's *referenced concept* and p is the attribute's *referenced property*. ATHENA maintains additional information about *attribute a*, such as aggregation functions and predicates applied, as well as its positions in the select, groupBy and orderBy clauses. The *join conditions* are constructed based on the binExpr2 expressions of the OQL query and show how the concepts referenced in the query are connected in the ontology. The *join condition* $j = (c_i, alias_i, c_j, alias_j, r)$ refers to the concepts c_i (through the alias $alias_i$) and c_j (through the alias $alias_j$), and to the relation r that connects them. For example, the query of Figure 5(a) contains a *join condition* between the oInvestment and oInvestorCompany aliases, through the investedBy relation.

The *attributes* and *join conditions* have an associated type, namely *concrete* and *virtual*. This type distinction is crucial since it allows our algorithm to handle various physical data representations. The *concrete* type implies that there is a direct mapping between the ontology elements referenced by the *attribute* or *join condition*, and the database objects of the relational schema. The *virtual* type, on the other hand, implies that there is a complicated relationship between the ontology elements and the actual physical representation. An *attribute a* is *concrete*, if the function $\overrightarrow{F}(\texttt{c.p})$ points to a database field that belongs to table $\overrightarrow{T}(c)$ and it is *virtual* when $\overrightarrow{F}(\texttt{p})$ is not defined (Section 3.1.3 describes when this is possible). For example, the *attribute* associated with the InvestorCompany.name property is *concrete* on the *denormalized* schema presented above, since the property is mapped to the RInvestorCompany.name field. However, it is *virtual* on the *normalized* schema because in this case, the RInvestorCompany table does not contain a name field. A *join condition* j is *concrete*, if the set $\overrightarrow{S}(r)$ contains only one FK constraint between the tables $\overrightarrow{T}(c_i)$ and $\overrightarrow{T}(c_j)$ and thus they can be joined directly. If more than one FK constraints are involved, the *join condition* is considered *virtual*. This typically happens when r is a many-to-many relation. In this case, a typical approach is to create an auxiliary table that breaks the relation into two one-to-many relations.

A query that consists of **only** *concrete attributes* and *join conditions* can be directly translated into a SQL query. This is because, the only database tables accessed are the ones that correspond to the concepts explicitly referenced in the sets of *attributes* and *join conditions*. For example, the OQL query of Figure 5(a) can be translated to a SQL query over the *denormalized* schema without further processing (Figure 5(b)). As we show next, the same OQL query needs further processing over the *normalized* schema.

3.3.2 Translation Algorithm

We now present an overview of our translation algorithm. Since an 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 (*QuerySet*). It also maintains the set of their corresponding SQL queries (*ResultSet*). The algorithm operates on each query of the *QuerySet*, performing *attribute* and *join condition transformations* until all the *virtual* elements of the query become *concrete*. The query is then converted into an equivalent SQL query. The transformations take a *virtual* element as input and produce new elements (*concrete* or *virtual*).

In a *virtual attribute*, the *referenced property* is not represented as a field in the database table that corresponds to the *referenced concept*. The *attribute transformations* derive the property from another concept that is connected with the referenced concept through an *inheritance* (*inheritance transformation*) or through a *membership* relation (*membership transformation*).

In case the referenced concept is a child or a member concept, the inheritance and membership transformations attempt to derive the referenced property from its corresponding parent or union concept, respectively. For example, the InvestorCompany.name and InvesteeCompany.name properties will be retrieved from the Company concept, in the *normalized* schema of our example. Both transformations replace the current *attribute* with a new one that still refers to the property but references the parent or union concept, through a new alias. The new alias guarantees that the query remains semantically correct, in case the concept is already referenced in the OQL query. In Figure 5(c), the application of two inheritance transformations introduced the RCompany table with two different aliases in the query, even though the Company concept was not present in the original OQL query. A new *join condition* between the parent and child (or union and member) concepts is also created using the appropriate aliases (e.g, a *join condition* between the Company and InvestorCompany concepts). This *join condition* ensures that the connection to the originally referenced child or member concept, is still maintained.

If the referenced concept is a union concept, the referenced property is derived from the properties of the member concepts. The membership transformation replaces the current query in the *QuerySet* with a set of new queries, each one referencing a member concept instead of the union concept.

The *join condition transformation* is applied when the two tables that correspond to the referenced concepts cannot be joined directly but an auxiliary table is involved in the join path. The transformation creates a dummy concept to represent the additional table, and breaks the original *join condition* into two *concrete join conditions*.

When the query consists of only *concrete* elements, the corresponding SQL query is generated, and is added to the *ResultSet*. After all the queries in the *QuerySet* have been translated, the algorithm generates the final SQL query by applying the SQL UNION operator on the queries of the *ResultSet*.

3.4 Handling Nested Sub-queries

ATHENA is able to handle more complex natural language queries that get translated into nested SQL queries. We now provide a high-level overview of how our system processes this type of queries.

After an NLQ has been annotated by the Ontology Evidence Annotator, ATHENA invokes the *Nested Query Detector* which examines whether the NLQ should be translated into a nested OQL query. This component employs a collection of lexicon-based techniques to decompose the input NLQ text into two sentences, namely the left-hand side (LHS) sentence corresponding to the outer OQL query, and the right-hand side (RHS) sentence corresponding to the inner OQL query. The inner query essentially represents a predicate on a property referenced in the outer query. At the next step, the Ontology-driven Interpretations Generator is invoked on the LHS and RHS sentences, and then the Ontology Query Builder generates two OQL queries corresponding to the two sentences. The inner query is incorporated in the `having` clause of the outer query, if it represents a predicate applied on an aggregation over a property of the outer query. Otherwise, it is incorporated in the `where` clause as a `binExpr1` expression. After the nested OQL query is generated, the Query Translator converts it into a nested SQL query. The details are ommitted in the interest of space.

Our current implementation supports single level nesting in the OQL queries. In our experiments with real queries, we rarely encountered a case where a deeply nested OQL query needs to be generated. This is because users tend to form simple NLQs. However, the OQL grammar can support arbitrary levels of nesting and we plan to extend our implementation to also cover these cases.

4. EXPERIMENTAL EVALUATION

In this section, we provide a comprehensive evaluation of ATHENA using various workloads. We also compare our system with NaLIR [19] and PRECISE [30], two state of the art systems that expose a natural language interface over relational databases.

4.1 Experimental Setup

We first describe the workloads used in our experiments, followed by our evaluation metrics.

4.1.1 Workloads

To evaluate the effectiveness of our system on different application domains, we use three workloads over geographical, academic and financial data described below:

- **GEO Workload**: The GEO workload is a popular workload that has been used in prior work [9, 33], including PRECISE [30]. It consists of 250 natural language queries over geographical data about United States. The data is stored in a relational database and the schema is well-specified.
- **MAS Workload**: The MAS workload is generated based on the Microsoft Academic Search dataset[3] which contains bibliographic information for academic papers, authors, journals, conferences, and universities. The workload was designed by the authors of [19], and consists of 196 natural language queries that have been used to evaluate the effectiveness of the NaLIR system. The relational schema corresponding to this dataset has been provided to us by the authors of [19].
- **FIN Workload**: The FIN workload is created by IBM based on real-world, financial data from various sources. The dataset is described by a complex financial ontology that contains 75 concepts, 289 properties and 95 relations. For this experiment, we conducted a user study. Ten IBM employees that have not used this dataset before, generated 108 NLQs against the ontology using ATHENA. The users observed ATHENA's output and marked whether it was able to find a correct interpretation as well as the rank of the correct interpretation. The results that we report are based on the users' responses. The dataset is stored in a relational database with a fully normalized schema.

[3] `http://academic.research.microsoft.com/`

	Ontology Characteristics				
Workload	$\|C\|$	$\|P\|$	$\|R_F\|$	$\|R_I\|$	$\|R_M\|$
GEO	14	20	15	0	0
MAS	17	23	20	0	0
FIN	75	289	55	34	6

Table 1: Summary statistics of the three ontologies

To evaluate ATHENA with the GEO and MAS workloads, we manually created the ontology and the corresponding Ontology-to-Database Mapping based on the provided database schema. Table 1 shows the characteristics of the three ontologies. As shown in the table, the FIN ontology is significantly more complex than the other two ontologies. It consists of a much larger number of concepts (C), properties (P), and functional relations (R_F) and it is the only one that contains inheritance (R_I) and membership (R_M) relations. To the best of our knowledge, our work is the first one to evaluate NLQs over relational databases using such a complex domain specific ontology and underlying relational schema.

The *dictionary* of synonyms for the elements of the GEO and MAS ontologies was manually populated. The synonyms for the FIN ontology were generated both manually and semi-automatically.

4.1.2 Evaluation Metrics

We have two objectives in our experiments. First, given a workload, we evaluate how often our system is able to find the correct NLQ interpretations, and thus returns correct results. We use the **precision** and **recall** measures to evaluate our system with respect to this objective. Second, we evaluate how often user interaction is required. Note that our system ranks the NLQ interpretations and presents to the user. The results of the SQL query corresponding to the top-ranked interpretation are automatically returned to the user. In case the top-ranked interpretation is not the correct one, the user can interact with the system and pick an alternative interpretation. We'd like to minimize user interaction as much as possible. We use the **mean reciprocal rank** to evaluate the effectiveness of our ranking scheme. The definitions of our three metrics are as follows:

- **Precision**: The precision is the number of NLQs correctly answered by the system divided by the total number of NLQs for which the system produced an answer.
- **Recall**: The recall is the number of NLQs correctly answered by the system divided by the total number of NLQs in the workload.
- **Mean Reciprocal Rank (MRR)**: Given that the system correctly answers N NLQs, its mean reciprocal rank is a standard statistic measure defined as:

$$\text{MRR} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{\text{rank}_i} \quad (1)$$

where rank_i refers to the rank position of the correct interpretation for the i^{th} NLQ. Ideally, the correct NLQ interpretation should be always the top-ranked one, since in this way no user interaction is required. Thus, the closer the value of *MRR* is to 1, the more effective the interpretation ranking scheme will be.

4.2 Experimental Results

Table 2 shows our experimental results. ATHENA achieves high precision and recall on all workloads. Moreover, the MRR values

show that ATHENA's interpretation scheme is very effective and thus the system requires user interaction only in some rare cases.

Workload	Precision	Recall	MMR
GEO	100%	87.2%	1.00
MAS	100%	88.3%	1.00
FIN	99%	88.9%	0.99

Table 2: ATHENA's performance on the three workloads

4.2.1 GEO Workload

As noted in Section 4.1.1, the GEO workload has been used to evaluate the PRECISE system [30] on geographical data. In this experiment, we evaluate ATHENA using the 250 queries of the GEO workload and compare its performance with PRECISE.

We now compare the two systems using the precision and recall numbers for PRECISE from [30]. Both systems achieve 100% precision. However, ATHENA is able to correctly answer more NLQs than PRECISE. More specifically, ATHENA achieves 87.2% recall whereas PRECISE achieves 77.5% recall. ATHENA never requires user feedback to find the correct interpretation for a NLQ. Its MMR value is 1 which denotes that when the correct interpretation is found, it is always the top-ranked one. The MRR metric is not applicable for PRECISE since the system does not rank the NLQ interpretations but instead returns the results of all the interpretations to the user. Note that PRECISE classifies some NLQs as *semantically intractable* and automatically rejects them. These NLQs contain tokens that cannot be mapped to any database object. ATHENA, on the other hand, is able to answer questions that are considered *semantically intractable* by PRECISE due to the the large set of synonyms associated with the ontology elements.

ATHENA correctly answers 218 out of the 250 NLQs of the GEO workload. For the remaining 32 NLQs, ATHENA cannot generate any interpretation. We identified three reasons for this behavior. First, 30 NLQs contain tokens that cannot be mapped to an ontology element. For example, the token "USA" in the NLQ *"What is the highest point of the USA?"* cannot be mapped to any of the ontology elements. This is because the GEO dataset does not contain the data value "USA", and thus the value is not indexed in the TI. A similar problem exists with the token "density" in the NLQ *"What state has the largest population density?"*. Note that we cannot fix this problem by adding more synonyms in our *dictionary* since the ontology does not contain any element that can qualify as a synonym for the "density" token. Second, the NLQ *"What rivers do not run through Tennessee?"* cannot be handled since our current implementation of the NLQ Engine does not support negation. Finally, ATHENA cannot produce any interpretation for the query *"Which states have points higher than the highest point in Colorado?"*. This NLQ implicitly references the `State.highestElevation` property of the GEO ontology and ATHENA is not able to make this association.

In some cases, ATHENA generates multiple top-ranked interpretations which all have the same score. For example, the NLQ *"How many people live in Washington?"* produces two top-ranked interpretations. In the first interpretation, the token "Washington" refers to the state of Washington, and in the second one it refers to the city of Washington. Both interpretations are correct and thus ATHENA presents the results of two SQL queries to the user.

4.2.2 MAS Workload

In this experiment, we evaluate ATHENA using the MAS workload that was also used to evaluate the NaLIR system [19]. As shown in Table 2, ATHENA is able to correctly answer 173 out of the 196 NLQs of the workload, achieving 88.3% recall and 100% precision. ATHENA is able to produce the correct interpretations without requiring any user feedback. More specifically, the correct interpretation for the 173 answered NLQs is always ranked first, effectively producing an MRR value of 1.

An evaluation of NaLIR on this workload is presented in [19]. The authors did not use the precision and recall metrics. However, they presented an analysis on the number of failed queries. The authors performed a user study in which each of the 196 NLQs in the workload was evaluated by multiple users. Note that in our experiment, we evaluate each NLQ only once. This is because ATHENA does not require user feedback to generate the correct interpretation for this workload and thus it always produces the same result. Although the results presented in [19] are not directly comparable to ours, it is clear that NaLIR performs poorly without user feedback during the interpretation process. More specifically, NaLIR was able to answer correctly only 65.3% of the queries without user interaction. In case the users provided feedback, NaLIR answered correctly 89.8% of the queries. Note that ATHENA provides a similar recall without requiring user interaction.

ATHENA is not able to generate an interpretation for 23 NLQs due to unmatched tokens and some nested query cases that we do not handle yet. Here we present 2 examples. Consider the NLQ *"Return me the authors who have cooperated both with H. V. Jagadish and Divesh Srivastava"*. In this case, the token "cooperated" cannot be mapped to an ontology element and thus the NLQ interpretation process stops. In the NLQ *"Return me the papers written by H. V. Jagadish and Divesh Srivastava"*, the tokens "H. V. Jagadish" and "Divesh Srivastava" are both mapped to the `Author.name` property. This NLQ cannot be represented by a single interpretation tree since the tree would contain a cycle that starts and ends at the `Author` concept. The solution is to express this NLQ using a nested OQL query. However, the *Nested Query Detector* component does not currently support this type of NLQs.

4.2.3 FIN Workload

In this section, we present the results of our user study on the FIN ontology. As mentioned in Section 4.1.1, the FIN ontology is much more complex than the MAS and GEO ontologies. The corresponding SQL queries of this workload contain aggregations, joins, unions and nested subqueries among others.

The output of ATHENA for each NLQ is presented to the user that formulated the NLQ. The user verifies whether the interpretation that captures her intent is generated and also denotes the interpretation's ranking position. ATHENA generated at least one interpretation for 97 out of the 108 NLQs of the workload. However, it did not produce the correct interpretation for 1 of these queries, effectively achieving a precision of 99%. As shown in Table 2, ATHENA's recall is 88.9% since it correctly answers 96 out of the 108 queries. For 94 of the 96 correctly answered queries, ATHENA identified the correct interpretation as the top-ranked one. The correct interpretation for the remaining 2 queries is ranked in the second position. Thus, the MRR value is 0.99 for this workload.

It is worth noting that 70% of the FIN queries, refer to a parent or a union concept in the ontology. Note that systems such as NaLIR or PRECISE would not be able to handle these NLQs. This is because they operate directly on the database schema and thus do not exploit the rich semantics of the ontology (inheritance and membership relations). Note that NaLIR and PRECISE experimented with datasets that do not include this type of relationships.

ATHENA is not able to generate an interpretation for 11 NLQs. Here are 2 examples. Consider the NLQ *"Total capital of small vs. large companies by year and state"*. In this case, ATHENA

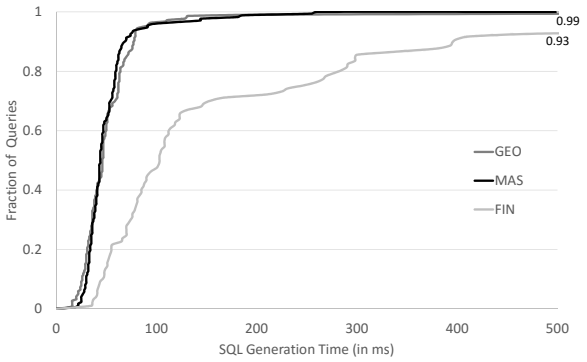

Figure 6: Cumulative distribution function of SQL generation time for the three workloads

does not understand the meaning of the tokens "small" and "large". In the case of the NLQ *"List the industry sector with highest ratio of the salary of CEO to the average salary of their employee"*, ATHENA does not understand the meaning of the token "ratio". The remaining NLQs were rejected for similar reasons.

As noted before, ATHENA produced an incorrect answer for 1 NLQ, namely *"In which sector did Citibank invest the most?"*. Our NLQ Engine creates a *selection set* that maps the token "sector" to the `Industry.sector` property and the token "Citibank" to the `InvestorCompany.name` property. As noted in Section 3.2.2, our interpretation algorithm finds the "simplest" interpretation that covers the *selection set*. In this case, ATHENA picked an interpretation tree that connects the `Industry` and `InvestorCompany` concepts through the `Company` concept. However, in the correct interpretation tree, these two concepts are connected through a longer path that spans four other intermediate concepts. Since, this interpretation tree is not the simplest one, ATHENA disregards it.

ATHENA ranks the correct interpretation of the following NLQ in the second position: *"Show me amount of transactions between Citibank and IBM"*. The *selection set* that forms the top-ranked interpretation maps both the "IBM" and "Citibank" tokens to the `Company.name` property and the token "amount of transactions" to the `Transaction.amount` property. The corresponding interpretation tree contains one edge that connects the `Company` and `Transaction` concepts, and thus has a score equal to 1. However, the correct interpretation is generated by another *selection set* in which the token "Citibank" is mapped to the `InsiderCompany.name` property. In this case, the interpretation tree is more complex and has a score equal to 2. Our NLQ Engine ranks the interpretation trees corresponding to different *selection sets* in increasing order of their score. Thus, the tree that corresponds to the correct interpretation is placed in the second position.

4.2.4 Performance Evaluation

In this section, we present an analysis of ATHENA's SQL generation time, i.e., the time spent in translating the input NLQ to the equivalent SQL query. In summary, for 90% of the NLQs contained in our three workloads, the system is able to generate the final SQL query in less than 120 ms, achieving its goal of interactive execution time.

Figure 6 shows, for each of the three workloads, the cumulative distribution function (CDF) of the SQL generation time, i.e., the fraction of queries that were translated in less than the time on the x-axis. We have excluded the queries for which ATHENA is unable to find the correct interpretation. To improve readability of the graph, we only show SQL generation times up to 500 ms. As shown in the figure, 99%, 100% and 93% of the queries in the GEO, MAS, and FIN workload respectively, are translated in less than 500 ms.

For the GEO and the MAS workloads, SQL generation completed within 200 ms for 99% of the queries. However, for the more complex FIN workload, 14% of the queries took longer than 300 ms to be translated. Breaking the SQL generation time down by components, we observed that a larger fraction of the time is spent in the Ontology-driven Interpretations Generator and Query Translator components for the FIN workload. We attribute this behavior to two factors.

First, many of the natural language queries in the FIN workload contain tokens such as "year" and "quarter", which match multiple time-related properties in the ontology. Thus, a much larger space of interpretations needs to be searched to identify the correct interpretation. Excluding queries with such time-related tokens brings the SQL generation time down in line with the other workloads (less than 200 ms each). In future work, we intend to further reduce the search space by heuristically filtering time-related properties before the interpretation tree is generated. Second, the large number of inheritance and membership relations in the FIN ontology forces the Query Translator to spend about 21 ms on average, as opposed to about 1 ms in the simpler GEO and MAS ontologies.

Overall, ATHENA demonstrates interactive SQL generation times for more than 90% of the queries in our three workloads.

5. RELATED WORK

The problem of designing natural language interfaces for databases (NLIDBs) has been studied extensively in the data management literature [3, 31]. To the best of our knowledge, our system is the first ontology-based NLQ engine for relational databases.

As was noted in [3, 19], early NLIDBs used grammars designed for specific databases, thus limiting their applicability to other databases. More recent works such as NaLIR [3] and PRECISE [30] operate directly on the database schema, as opposed to our two stage approach designed to exploit the powerful semantics of the ontology. While NaLIR relies on user interaction to find the correct interpretation for ambiguous NLQs, PRECISE returns all interpretations deemed correct according to its own approximate definition of correctness. In contrast, ATHENA uses a novel ranking scheme that greatly improves user experience, and as shown in our experiments, it gives results that are comparable, if not more correct, than both these systems. Nauda [16] is an early interactive NLIDB. However, it focuses on providing additional information than explicitly required by the NLQ with the goal of generating responses that are useful as possible (over-answering). CANaLI [24] answers NLQs over general-purpose knowledge bases using an RDF (as opposed to a relational) data model. However, it only supports controlled natural language queries (following a specific template defined by an automaton) and it does not consider inheritance and membership relations in the ontology.

Keyword search interfaces over relational databases is another well-studied area [2, 4, 6, 7, 8, 13, 14, 32, 34]. Generally, NLIDBs support a much richer query mechanism than a flat collection of keywords. However, the techniques developed for keyword search systems can be useful in the context of NLIDBs. As noted in Section 3.2.2, ATHENA builds on Steiner Tree based techniques that have been developed for keyword search systems but significantly extends them to handle the rich semantics of the ontology.

Works such as [10] study the problem of generating an ontology from a relational database. These works are orthogonal to ATHENA, since our system is flexible enough to operate on an ontology and the corresponding relational database as long as an ontology-to-database mapping is provided. However, these works can be very useful for users who have an existing database and want to generate a corresponding ontology and query it using ATHENA.

Similar to OQL, UNL [17] is an intermediate language that has been used in natural language applications. However, UNL [17] targets mostly multilingual translation, and thus does not operate on top of an ontology. As opposed to the work in [17], ATHENA translates the NLQs into SQL and not into another natural language. As a result, the OQL language constructs are designed to facilitate the conversion to SQL whereas the UNL hypergraphs do not have this functionality since it is not required for multilingual translation. Another important distinction is that an OQL query represents one or more trees of an ontology graph whereas the UNL hypergraphs [17] encode a natural language statement without associating with a particular ontology.

WordNet [25] and BabelNet [26] are two popular frameworks which characterize English words and provide sets of synonyms for these words. ATHENA would be able to leverage such frameworks to generate synonyms during its offline processing phase. However, a challenge with these frameworks is that they do not sufficiently incorporate the terminology of specialized domains such as finance, healthcare, etc. Consequently, they might not be able to cover specific domains adequately and thus they are often substituted by domain-specific dictionaries developed in house. ATHENA will easily be able to accommodate such frameworks once they are able to cover domain-specific applications.

6. CONCLUSIONS AND FUTURE WORK

We have presented ATHENA, an ontology-driven system for natural language querying of complex relational data stores. Using an ontology-driven approach has two main advantages. First, it enables a richer semantic basis for interpreting the NLQ as compared to approaches which employ the database schema directly as the basis. Our experimental results demonstrate the efficacy of our ontology-driven interpretation algorithm. Second, the ontology provides physical independence, by separating logical and physical schemas. This enables the user to operate on the semantic ontology level and allows the system to generate queries on different relational schemata, or possibly even beyond relational data stores.

We plan to extend ATHENA to include natural language dialogue capabilities such that user can ask followup questions using the context of the previous questions. This will also enable the user to get answers to complex queries by engaging in a natural language conversation with the system. Another extension is to handle self-join queries through nesting. We also envision to use the ontology's structure and its associated data for automatically generating meaningful query suggestions for auto-completion of user questions. Additionally, we plan to incorporate query recommendation techniques in ATHENA in order to help the users understand which questions can be answered from a particular ontology. Finally, we plan to extend ATHENA to support NLQs over graph and document stores, and eventually allow cross-store queries.

7. REFERENCES

[1] Weakly Connected Directed Graph. `http://mathworld.wolfram.com/WeaklyConnectedDigraph.html`.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System For Keyword-Based Search Over Relational Databases. In *ICDE*, 2002.

[3] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural Language Interfaces to Databases–An Introduction. *Natural Language Engineering*, 1(01):29–81, 1995.

[4] Z. Bao et al. Exploratory keyword search with interactive input. In *ACM SIGMOD*, 2015.

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

[6] S. Bergamaschi et al. QUEST: A Keyword Search System for Relational Data Based on Semantic and Machine Learning Techniques. *Proc. VLDB Endow.*, pages 1222–1225, 2013.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.

[8] L. Blunschi et al. SODA: Generating SQL for Business Users. *Proc. VLDB Endow.*, 5(10):932–943.

[9] P. Cimiano and M. Minock. Natural Language Interfaces: What is the Problem?–A Data-driven Quantitative Analysis. In *Natural Language Processing and Information Systems*, pages 192–206. Springer, 2010.

[10] C. Curino et al. Accessing and Documenting Relational Databases Through OWL Ontologies. FQAS, 2009.

[11] Stanford Dependency Parser. `http://stanfordnlp.github.io/CoreNLP/depparse.html`.

[12] I. Feinerer and K. Hornik. *WordNet Interface*, 2016.

[13] V. Ganti, Y. He, and D. Xin. Keyword++: A Framework to Improve Keyword Search over Entity Databases. *Proc. VLDB Endow.*, pages 711–722, 2010.

[14] V. Hristidis and Y. Papakonstantinou. Discover: Keyword Search in Relational Databases. In *VLDB*, 2002.

[15] L. Kou, G. Markowsky, and L. Berman. A Fast Algorithm for Steiner Trees. *Acta Informatica*, 15(2):141–145, 1981.

[16] D. Küpper, M. Storbel, and D. Rösner. NAUDA: A Cooperative Natural Language Interface to Relational Databases. In *ACM SIGMOD*, 1993.

[17] M. Lafourcade and C. Boitet. UNL Lexical Selection with Conceptual Vectors. In *LREC*, 2002.

[18] H. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Data Bases. In *SSDBM*, 1997.

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

[20] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: An Interactive Natural Language Interface for Querying XML. In *ACM SIGMOD*, 2005.

[21] D. Lin. Automatic Retrieval and Clustering of Similar Words. In *Proceedings of the 17th International Conference on Computational Linguistics - Volume 2*, COLING, 1998.

[22] W. S. Luk and S. Kloster. ELFS: English Language from SQL. *ACM Trans. Database Syst.*, 11(4):447–472, 1986.

[23] C. D. Manning et al. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL System Demonstrations*, 2014.

[24] G. M. Mazzeo and C. Zaniolo. Answering Controlled Natural Language Questions on RDF Knowledge Bases. In *EDBT 2016*, pages 608–611, 2016.

[25] G. A. Miller. WordNet: A Lexical Database for English. *Commun. ACM*, 38(11), 1995.

[26] R. Navigli and S. P. Ponzetto. BabelNet: The Automatic Construction, Evaluation and Application of a Wide-coverage Multilingual Semantic Network. *Artif. Intell.*, 193, 2012.

[27] T. B. Pedersen et al. Extending Practical Pre-Aggregation in On-Line Analytical Processing. In *VLDB*, 1999.

[28] J. Plesník. A bound for the steiner tree problem in graphs. *Mathematica Slovaca*, 31(2):155–163, 1981.

[29] A.-M. Popescu et al. Modern Natural Language Interfaces to Databases: Composing Statistical Parsing with Semantic Tractability. In *COLING*, 2004.

[30] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a Theory of Natural Language Interfaces to Databases. In *IUI*, 2003.

[31] U. Shafique and H. Qaiser. A Comprehensive Study on Natural Language Processing and Natural Language Interface to Databases. *International Journal of Innovation and Scientific Research*, 9(2):297–306, 2014.

[32] A. Simitsis, G. Koutrika, and Y. Ioannidis. PréCis: From Unstructured Keywords As Queries to Structured Databases As Answers. *The VLDB Journal*, pages 117–149, 2008.

[33] L. R. Tang and R. J. Mooney. Using Multiple Clause Constructors in Inductive Logic Programming for Semantic Parsing. In *ECML*. 2001.

[34] S. Tata and G. M. Lohman. SQAK: Doing More with Keywords. In *ACM SIGMOD*, 2008.

[35] TIMEX Annotator. `http://ilps.science.uva.nl/resources/timextag/`.