Ontology-Based Natural Language Query Interfaces for Data Exploration

Chuan Lei, Fatma Özcan, Abdul Quamar, Ashish Mittal,
Jaydeep Sen, Diptikalyan Saha, Karthik Sankaranarayanan
IBM Research - AI

1 Introduction

Enterprises are creating domain-specific knowledge bases by curating and integrating all their business data, structured, unstructured and semi-structured, and using them in enterprise applications to derive better business decisions. One distinct characteristic of these enterprise knowledge bases, compared to the open-domain general purpose knowledge bases like DBpedia [16] and Freebase [6], is their deep domain specialization. This deep domain understanding empowers many applications in various domains, such as health care and finance.

Exploring such knowledge bases, and operational data stores requires different querying capabilities. In addition to search, these databases also require very precise structured queries, including aggregations, as well as complex graph queries to understand the various relationships between various entities of the domain. For example, in a financial knowledge base, users may want to find out "*which startups raised the most VC funding in the first quarter of 2017*"; a very precise query that is best expressed in SQL. The users may also want to find all possible relationships between two specific board members of these startups, a query which is naturally expressed as an all-paths graph query. It is important to note that general purpose knowledge bases could also benefit from different query capabilities, but in this paper we focus on domain-specific knowledge graphs and their query needs.

Instead of learning and using many complex query languages, one natural way to query the data in these cases is using natural language interfaces to explore the data. In fact, human interaction with technology through conversational services is making big strides in many application domains in recent years [13]. Such interfaces are very desirable because they do not require the users to learn a complex query language, such as SQL, and the users do not need to know the exact schema of the data, or how it is stored.

There are several challenges in building a natural language interface to query data sets. The most difficult task is understanding the semantics of the query, hence the user intent. Early systems [3, 30] allowed only a set of keywords, which had 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, 24, 29] that require good training sets, which are hard to obtain. Others require user feedback [14, 17, 18]. However, excessive user interaction to resolve ambiguities can be detrimental to user experience.

In this paper, we describe a unique end-to-end ontology-based system for natural language querying over complex data sets. The system uses domain ontologies, which describe the semantic entities and their relationships, to reason about and capture user intent. To support multiple query types, the system provides a poly store

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

architecture, which includes a relational database, an inverted index document store, a JSON store, and a graph database. As a result, we support very precise SQL queries, document search queries, JSON queries, as well as graph queries.

Once the data is stored in the appropriate backends, we also need to provide the appropriate abstractions to query the data without knowing how data is stored and indexed in multiple data stores. 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 the second stage, each OQL query is translated into a backend query (e.g., SQL) by using the mapping between the ontology and the corresponding data schema of the backend.

In this two-stage approach, 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 the users' interaction for disambiguation. Using an ontology in the interpretation provides a stronger semantic basis for disambiguation compared to operating on a database schema.

OQL provides a powerful abstraction layer by encapsulating the details of the underlying physical data storage from the NLQ interpretation, and relieving the users from understanding what type of backend stores are utilized, and how the data is stored in them. The users only need to know the domain ontology, which defines the entities and their relationships. Our system understands the mappings of the various ontology concepts to the backend data stores as well as their corresponding schemas, and provides query translators from OQL to the target query language of the underlying system, providing physical independence.

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. The second step in the process translates OQL to the underlying target data store, using the ontology-to-schema mappings.

Conversational interfaces are the natural next step, which extends one-shot NLQ to a dialog between the system and the user, bringing the context into consideration, and allowing informed and better disambiguation. In this paper, after we describe our end-to-end NLQ system, we also discuss how to extend and adapt our ontology-based NLQ system for conversational services, and discuss the challenges involved.

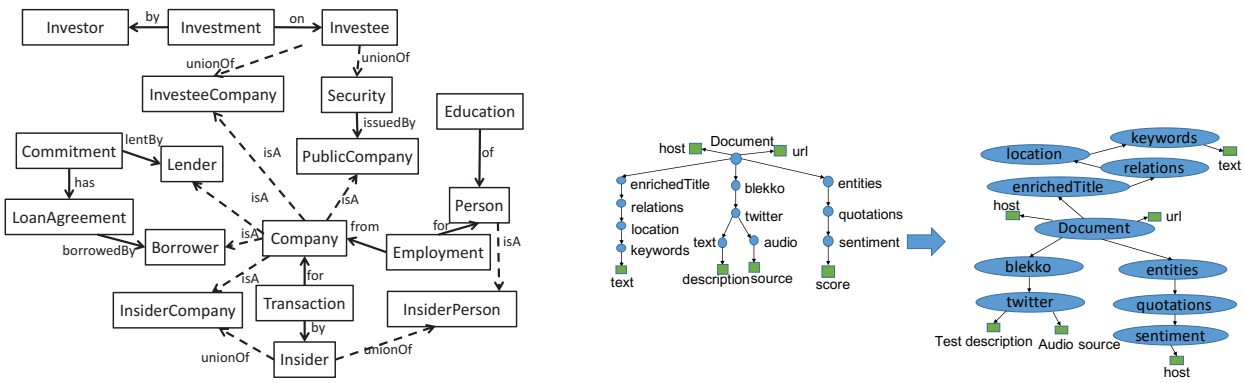
In the following, we first describe how we use ontologies, and the ontology query language (Sections 2 and 3), followed by the overall system description (Section 4). The translation index is used to capture the domain vocabulary and the domain taxonomies (Section 5). Section 6 discusses the details of the NLQ engine and Section 7 discusses the challenges in NLQ and conversational interfaces. We highlight some use cases where use our end-to-end system in Section 8, review related work in Section 9, and finally conclude in Section 10.

2 Ontology

2.1 Ontology Basics

The domain ontology is a core piece of technology, which is central to our system. It describes the domain and provides a structured entity-centric view of the data corresponding to the domain. Specifically, the ontology describes the entities relevant to the domain, the properties associated with the entities, and the relationships among different entities. We use OWL to describe our ontologies. It provides a very rich and expressive data model that can capture a variety of relationships between entities such as *functional*, *inheritance*, *unions*, etc. We use the domain ontology to just describe the schema of the domain capturing its meta-data. As such, in our proposed system, ontologies do not contain any instance data.

Figure 1(a) shows a small portion of an ontology that we use in a finance application [26, 8]. It includes concepts such as *Company*, *Person*, *Transaction*, *Loan agreement* etc. Each of these concepts are associated



(a) A semantic graph of partial finance ontology

(b) Ontology generation from schema

Figure 1: Sample Ontology and Ontology Generation

with a set of data properties: For example, the concept *Company* is associated with *name*, *stock symbols*, etc. Object properties represent the associations between the concepts. For example, *Securities* are *issued by* a *Public Company* where *issued by* is an object property. In this paper, we will use the extended version of this financial ontology to illustrate system functionality.

2.2 Generating Ontologies from Data

There are many cases where the ontology is provided, especially if the data is curated, but there are also cases where we only have the data without a corresponding ontology. To provide NLQ over such data sets, we provide techniques to infer the ontology, i.e., the concepts and their relationships relevant to the domain, from the underlying data and its schema.

Ontology generation from JSON data. This process involves the discovery of the domain ontology from all the JSON documents stored in a document store. This is a multi-step process. The first step involves the extraction of the schema tree from the nested JSON structure of each individual document. These individual schema trees are merged into a single schema, similar to the data guide generation process of Goldman et al. [11]. In the second step, we use the following rules to convert the schema into an ontology.

Path $A.b \Rightarrow$ Concept A, Property b of A

Path $A.B.c \Rightarrow$ Concept A, Concept B, Relation A to B, Property c of B

The ontology to physical schema mappings are an essential part of query translation, and are generated as part of ontology discovery. Figure 1(b) shows the schema that is generated by our techniques and the ontology generated from the schema, respectively.

Ontology generation from relational databases. Various ontology elements can be inferred from an RDBMS. *Table Inference.* Ontology supports two different types of relations - functional and ISA. The table inference is nontrivial as they depend on the Unique (Primary) Key and Foreign Key interactions, and quite often these keys are not specified in the database, especially when the database is created from raw data files. The steps are as follows:

- Identifying Unique Key and Foreign keys: For each table, we identify unique keys by comparing the count of the distinct values of the column and the total row count in the table. If they are identical, we assert a unique key constraint. Similarly, for foreign key, we check if the rows in the join of the two tables based on the selected columns are equal to the total rows of the referring table.

- Inferring ISA relation: Once all the unique keys and FKs have been derived, we find all the tables having a single column, which is both a unique key and the FK for the table. In such a case, we assert an ISA relation

between this table and the referred table. There might be spurious ISA relation if there are auto-incremented columns in two otherwise unrelated tables. To eliminate such cases, we check if the values in the column of the referencing table are consecutive integers. The rest of the tables are deemed functional.

- Inferring cardinality of functional relations: For inferring m:n relations, we find tables with exactly two columns, and both acting as foreign keys to different tables in the data base. If such a case occurs, the two tables that are being referred to are mapped in an m:n relation and the join table does not take part in any relation. We identify the relations with unique values in the FK and mark it as 1:1, and the rest as 1:n.

Concepts and Properties. Each table is mapped to a concept except the ones that form the join tables for the m:n relations, and all columns except the foreign keys in that table map to the data properties of that concept. The data type of the properties are also extracted from the data type of the table columns. The resulting ontology is stored in OWL2 [1] format.

3 Ontology Query Language

We propose OQL, a query language expressed against the domain ontology as an abstraction to query the data without knowing how data is stored and indexed in multiple data stores. OQL represents queries that operate on a set of concepts and relationships in an ontology. OQL is inspired by SQL and its object-oriented extensions, and it provides very similar constructs, including SELECT, FROM, WHERE, HAVING, GROUP BY and ORDER BY clauses. As such, OQL can express sophisticated queries that include aggregations, unions, nested sub queries, document, fielded search over document stores, as well as JSON paths, etc.

OQL allows several types of predicates in the WHERE clause including: (1) predicates used to compare concept properties with constant values (or sets of values), (2) predicates used to express joins between concepts, (3) predicates that use binary operations such as *MATCH* for full text search or fielded search over documents, and (4) predicates that use path expressions to specify the application of a predicate along a particular path over the domain ontology. In general OQL is composable and allows arbitrary level of nesting in the SELECT, FROM, WHERE and HAVING clauses.

Example 1: Show me all loans taken by Caterpillar by lender. This query joins the following concepts in the ontology: *lender*, *borrower*, *commitment* and *loanAgreement*. In this query, Caterpillar is the borrower. The condition *br = la→borrowedBy* indicates that the borrower instance should be reachable by following the *borrowedBy* relationship from the *LoanAgreement* concept.

```
SELECT   Sum(la.amount)
FROM     Lender ld, Borrower br, Commitment c, LoanAgreement la
WHERE    br.name IN ('Caterpillar', 'Caterpillar Inc.') AND ld = c→lentBy
         AND la = c→has AND br = la→borrowedBy
GROUP BY ld.name
```

Example 2: The following query applies a fielded search using the *MATCH* binary operation for a person whose name contains the word *Adam* and has held the position/title of *President* in a company.

```
SELECT  oPerson.name, oEmployment.position, oEmployment.title, oCompany.name
FROM    Person oPerson, Employment oEmployment, Company oCompany
WHERE   oPerson.name MATCH ('Adam') AND oEmployment.title='PRESIDENT'
        AND oEmployment→for=oPerson AND oEmployment→from = oCompany
```

4 System Architecture

Figure 2 shows the overall system architecture. It serves a wide variety of applications including NLQ Service, Conversational Services that enable chat bots, and other applications that use programmatic APIs. These ap-

plications use OQL queries to explore and analyze the underlying data, stored in one of the supported backend stores.

The *OQL query compiler* and the *OQL query translators* together translate the OQL query into the target back-end query using the ontology to physical schema mappings. By decoupling the semantic domain schema from the actual data storage and organization, our system enables independent optimization of each layer, and allows the applications to reason at the semantic level of domain entities and their relationships.

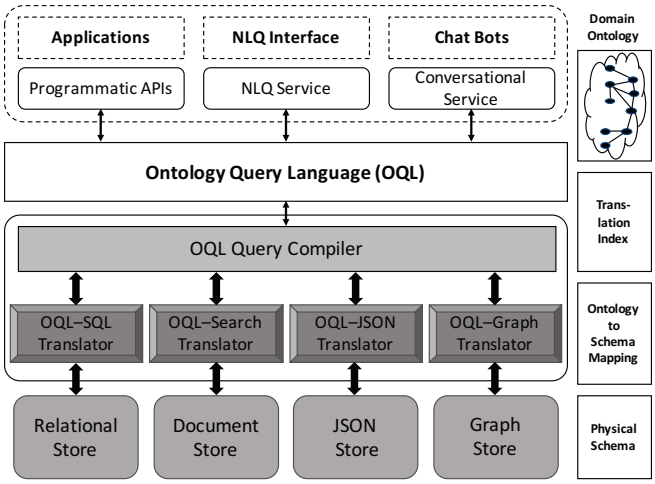


Figure 2: System Architecture

OQL Query Compiler. The OQL query compiler includes an OQL query parser, and a set of query translators one for each back-end. The query parser takes an OQL query as input and generates a logical representation of the query in the form of a Query Graph Model (QGM) [22]. Using QGM as the logical representation of the query defers the choice of the actual physical execution plan to the underlying data store that would be responsible for the execution of the query. The query compiler identifies which backend store contains the data needed by the query, and routes the query to that target. It may also generate a multi-store execution plan, which is optimized to minimize data movement between data stores.

OQL Query Translation. The query fragments expressed over the ontology are translated into appropriate back-end queries to be executed against the physical schema of the stored data. We have developed several query translators (one per type of back-end store) for this purpose. Query translation requires appropriate schema mappings that map concepts and relations represented in the domain ontology to appropriate schema objects in the target physical schema.

Query translation also needs to handle *union* and *inheritance* relationships, and translate path traversals to a series of join conditions. Depending on the physical data layout these are translated to appropriate operations supported by the back-end stores. For document oriented stores the translator needs to make distinctions between fielded and document oriented search. For querying JSON data it needs to understand where to apply predicates along specific paths in the nested schema.

5 Translation Index

Translation Index (TI) is a standalone component that captures the domain vocabulary, providing data and metadata indexing for data values, and for concepts, properties, and relations, respectively. TI helps the NLQ engine to identify the entities in the query, and their corresponding ontology properties.

TI captures both internal as well as external vocabularies. For the internal ones, TI is populated from the underlying data stores during an offline initialization phase. For example, if the input query contains the token "Alibaba", TI captures that "Alibaba" is a data value for the name column in the Company table in the relational back-end store. Note that "Alibaba" can be mapped to multiple ontology elements (e.g., InvestorCompany or Lender), and TI captures all of them. TI provides powerful and flexible matching by using semantic variant generation schemes. Essentially, for the data values indexed in TI, we not only index the actual values (e.g., distinct values appearing in Company.name), but also variants of those distinct values. TI leverages semantic variant generators (SVGs) for several common types, including person and company names, among others. For example, given an input string "Alibaba Inc", the company name SVG produces the following list of variants:

{“Alibaba”, “Alibaba Inc”, “Alibaba Inc.”, “Alibaba Incorporated”}. This allows the NLQ engine to formulate the queries by using any of the indexed variants of a data value (e.g., “Alibaba” vs. “Alibaba Inc”), or use these indexed variants in an IN (list) predicate.

Internal vocabulary is not always sufficient. Suppose “renal impairment” is a data value in the back-end store. If the query term is “kidney disease”, TI would not be able to return “renal impairment” even though “renal impairment” is a more specific term for “kidney disease”. To address this issue, TI also leverages external knowledge sources, such as standard taxonomies, domain lexicons, dictionaries, etc. To continue our example of searching for “kidney disease”, TI uses SNOMED [2], a systematically organized taxonomy of medical terms, to connect “kidney disease” with “renal impairment” via the *ISA* relationship in SNOMED. Hence with more domain knowledge, the semantically related matches can be found by TI, improving recall for the query.

6 Natural Language Querying

In this section, we describe how the *NLQ Engine* takes a natural language query (NLQ) and interprets it over the domain ontology to produce an OQL query. The transformation of NLQ to OQL query happens in multiple phases. We describe each of the phases with a running example from the finance domain as captured in Figure 1(a) for natural language query: *“Show me total loan amount given by Citibank to Caterpillar in 2016”*.

Evidence Generation. In the very first step, *NLQ Engine* tries to identify all the different mentions of ontology elements in the NLQ. So the algorithm first iterates over all the word tokens in the NLQ and collects *evidences* of one or more ontology elements (*concept*, *relation*, and *property*) which have been referenced in the input NLQ.

In general, a token can match multiple elements in the ontology. For example, the token “Citibank” is mapped to *Lender.name* and *Borrower.name* (considering Citibank may have acted as a borrower as well as a lender) and the phrase “loan amount” is mapped to *LoanCommitment.amount*. The phrase “given by” gets mapped to the *lent by* relation in the ontology. “Caterpillar” is also mapped to *Lender.name* and *Borrower.name*. The temporal expression “in 2016” gets mapped to *LoanCommitment.year*.

An evidence can be of three types: 1) a *metadata* evidence is generated by matching the *synonyms* associated with the ontology elements (e.g., “loan amount”), 2) a *data-value* evidence is generated by looking up a token in the *Translation Index* (e.g., “Citibank”,”Caterpillar”), and 3) a *typed* evidence is generated by recognizers such as temporal expressions that recognize date/time related expressions and maps them to date/timestamp typed properties in the ontology. For example, “in 2016” is detected and mapped to *LoanCommitment.year*. The evidence for a token can either be *metadata* or *data-value*, but not both. Note that if a token matches a data-value corresponding to an inherited property of a child concept then it should also match with the same property of the parent. For example, if “Citibank” data-value matches *Lender.Name* then it should also match *Company.Name*.

The metadata matching uses synonyms associated with the ontology elements. The NLQ engine relies on the given set of synonyms associated with ontology elements. It also uses publicly available synonym databases such as PPDB [10] to create additional synonyms for the given set of terms.

Interpretation Generation. Note that, only one element from the evidence set of each token corresponds to the correct query. In this phase, the NLQ engine tries all such combination of elements from each evidence set. Each such combination, called *selection set*, is used to generate an *interpretation*, which is represented as a subgraph in the semantic graph connecting one evidence for each token respecting ontology-related constraints. This semantically grounds the words in the natural language to specific meanings by referring to elements in the semantic graph. Connecting these referred elements produces a unique interpretation for the given natural language query based on the ontology semantics.

For each selection set, a subtree, called Interpretation Tree, is computed (if possible) which uniquely identifies the relationship paths among the evidences in the selected set. It is computed by connecting all the elements in the selected set in the semantic graph and satisfying two constraints. One such constraint is the *relationship*

constraint which asserts that between evidence mapping "loan amount", "given by", and "Citibank" there should be a path in the interpretation tree connecting them in order. When "Citibank" is mapped to *Borrower* then this constraint will not be satisfied. The other constraint is called inheritance constraint. Inheritance constraint invalidates those interpretation trees which have a parent (or union) concept as a chosen element and it connects to a property or a relation which belongs to one of its child (or member) concepts in the ontology.

Note that many interpretations can arise since there can be many possible subtrees connecting elements of a selected set in addition to many possible selected sets. We use a Steiner-tree-based algorithm to generate a single interpretation of minimal size from a selected set. The rationale of computing Steiner trees to derive the minimal sized connected tree follows from the intuition that among the different possible subtrees, the most compact one has the most direct and straight forward meaning in the ontology and thus is the most likely intent of the user query. Following the same intuition, it also employs a ranking criteria to choose one or more interpretations across all selected sets depending on the number of edges present in the minimal Steiner tree computed from a specific selected set. Fewer edges in the Steiner tree results into higher rank of the interpretation. For example, the top ranked interpretation as found from selected set is *ITree* = {(*LoanCommitment→lent by→Lender*), (*LoanCommitment→has→LoanAgreement*), (*LoanAgreement→borrowed by→Borrower*)}. Note that, the ranking can result in multiple interpretations with the top rank score, if there are multiple minimal Steiner trees with the same number of edges. In that case NLQ will provide multiple answers to the user as obtained from multiple interpretations for the question.

OQL Query Generation. 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 the final step, 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 is formed by the evidence concepts in the interpretation graph. The path between the root node to other evidence nodes in the tree denote the join path between the concepts. Thus, the FROM clause for the example *ITree* will be *FROM Lender LenderObject, Borrower BorrowerObject, LoanCommitment LoanCommitmentObject*.

- WHERE clause. The WHERE clauses are generated based on the filter conditions specified. For example, to generate equality condition in the WHERE clause, data value evidences are used to determine the equality between the ontology property and the original value in the data value evidence. We uses a large vocabulary (e.g., more than, less than, etc.) for creating expressions. We also employ a grammar to recognize time-related expressions. For example, the WHERE clause corresponding to the example *ITree* will be *WHERE LenderObject.name = 'Citibank' AND BorrowerObject.name = 'Caterpillar' and LoanCommitmentObject.year = 2016*.

- 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. The OQL query generation algorithm identifies explicit references to aggregation functions (e.g., SUM/total) in the NLQ by employing a lexicon of terms corresponding to the common aggregation functions. In this example, the expression, *SUM(LoanCommitmentObject.amount))*, will be created. The display/non-aggregation properties of the SELECT clause are generated by finding ontology properties which appear after the head phrases such as "Show me", "Tell me" and not associated with another keyword such as "per" which can signify GROUPBY clause.

- GROUPBY clause. It specifies an ordered list of ontology properties that the user wishes to group the results by. A group by operation is recognized by presence of specific keywords like "by" , "for each" etc. followed by an ontology property.

- 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. Like group by, order by operation is also recognized by presence of specific keywords like ”order by”, ”sort by” followed by a property in the ontology.

7 Challenges and Vision

In this section we will discuss some of the common challenges encountered by the *NLQ Engine*. These are still open problems where active research is being done. Then, we also present our vision on how *NLQ Engine* can be extended to create a *Conversational System* that can resolve ambiguity via user interaction, which is also one of the core challenges in *NLQ Engine*.

7.1 Ongoing Work

Spurious Mapping in Evidence Generation. Consider an example query in “Show me companies Caterpillar has borrowed money from”. In this example, the token “money” happens to match a person name in the underlying data store and therefore associates the candidate *Person.Name* property to that token. Such mapping is an instance of spurious mapping. Detecting and handling spurious mapping remains a challenge in developing robust interpretation generation algorithms. A possible way to do this is to use the contextual knowledge from the input query to identify the query is asking about “Loans” and “Loans” does not involve “Person” as per the ontology model. This knowledge can be utilized to prune out such spurious mappings from being included in the interpretation process.

Nested Query Handling. Although OQL grammar supports multiple levels of nesting, detecting an NLQ which is to be translated to a nested OQL query and subsequently producing correct OQLs for it remains an open challenge. A part of the difficulty lies in taking clues from the natural language for detecting different cases of possible nesting. A harder challenge is to identify the sub-queries and the correct join condition to stitch the results of the sub-queries. Although this still remains an open problem, here we try to provide a high-level overview of our ongoing effort to address this problem. We employ a nested query detector, which builds on different lexicon rules to detect different cases of nesting. For each of the different cases, a reasoning engine can work towards building different sub-queries. Consider an example question as “show me all IBM transactions with value more than average selling price”, this can be detected as a nested query by the presence of comparator phrases like “more than” in the input query and a comparison against another aggregation value like “average selling price”. A possible way to handle this query is to decompose the input NLQ text into two sentences across the comparator phrase i.e., LHS sentence as “show me all transactions with value” which corresponds to the outer OQL query, and RHS sentence “average selling price” which corresponds to the inner OQL query. The inner query, in this example, represents a predicate on a property referenced in the outer query i.e., Transaction.amount. At the next step, the Ontology-driven Interpretations Generator can be invoked on the LHS and RHS sentences, and then the Ontology Query Builder can generate two OQL queries corresponding to the two sentences. The reasoning engine can figure out the correct join between the inner and outer query result. For example, if they are aggregation comparisons on both sides, then the inner query can be incorporated in the HAVING clause of the outer query, or else if it is a property value comparison, like in this example question, then a simple property value comparison in the WHERE clause is performed between outer query and inner query result; e.g., $Transaction.amount > Average(Price)$.

7.2 Extending NLQ for Conversation

Typical natural language interfaces are stateless and so they are oblivious to the history of the questions being asked in the past. Users, while asking questions to the NLQ interfaces, are latently aware of the context and ask questions keeping that in mind. Consider the following interaction in the finance domain - (i) *What is the*

loan amount given by Citibank to Caterpillar in each year? (ii) *Who are its lenders?* Here, the second question is dependent on the context of the first question. These are natural interactions for users. The conversational interface enables users to automatically exploit the latent semantic context of the conversation, thereby making it simpler to express complex intents in a natural, piece-wise manner.

There are several challenges in supporting such conversational interfaces. These challenges arise in interpreting the contextual questions in the presence of the context. In the subsequent subsections, we list the challenges and how they are supported via extending the Natural Language Querying described in section 6.

Co-reference Resolution. In a natural language dialog, users often use co-references to refer to the entities in the previously asked question. These co-references are typically pronouns, anaphora, cataphora, and sometimes split antecedents. There are many state of the art solution for co-reference resolution, but all of them are trained on general purpose news data [15, 19, 21, 23]. They fail to identify the subtleties of any specific domain because of the lack of training data.

For example a follow up query - "*Who are its lenders?*" to the base query - "*What is the loan amount given by Citibank to Caterpillar?*", contains a pronoun *its* that needs to be resolved. There are two candidate to *its* viz. (i) *Citibank* (the lender) and (ii) *Caterpillar* (the borrower). Any state of the art system will fail the subtle difference between the companies, which are taking different roles in the domain. To circumvent this weakness, we use signals from the Ontology to resolve co-references. In this particular case, from the ontology described in Figure 1, a borrower can only have lenders (determined through path between the concepts), and hence correct resolution will associate the co-referent *its* to Caterpillar. Thus, using domain specific ontology augments existing techniques and yields improved accuracy.

Ellipsis Resolution. Another challenge in conversational interface is to support ellipsis (Non Sentential Utterances) [9]. Ellipsis are partial questions that modify the previously asked question. A sample interaction involving ellipsis is as follows - (i) "*What is the loan amount given by Citibank to Caterpillar?*", (ii) "*in 2015?*" Here, the second question cannot be interpreted on its own and modifies the previously asked question. In this particular case, a WHERE clause on the loan period should be added to the previous query.

To resolve the ellipsis, we use the Ontology to find the annotations (select, where) in the follow up query and then use these annotations to apply transformations to the previous query. For the example described above, the annotations in the base query are - *loan amount* is part of SELECT clause, *Citibank* and *Caterpillar* are part of WHERE clause. This process is described in detail in section 6. For the follow up query, we identify the temporal condition as a WHERE clause predicate, and since there is no existing temporal clause in the previous query, we add it to the previous query.

Disambiguation. Often times in natural language queries, users employ terms that are ambiguous, e.g., *Southwest* as (i) *Southwest Airlines* or (ii) *Southwest Securities*. A conversational interface can help resolving these ambiguities via interaction in natural language. Sometimes these ambiguities can be automatically resolved if the users have mentioned the unambiguous choices in the previous questions. If no such clarity is available from the context, the interaction proceeds by asking the user to clarify.

Context Management. Since a conversational interface needs to maintain the context of the previously asked questions and answers, the interface can no longer be stateless. Context manager is responsible for updating the context as conversation proceeds. The conversation is initialized with an empty context. When user asks a question, all the information related to that question(including annotations, answers, etc.) are packaged as a context object and pushed to the context, which is represented as a stack. If additional user input is required for disambiguation, a new context state is created specifically for the disambiguation. Once an answer to disambiguation question is detected, the entire disambiguation object is removed from the context, and the last question that caused the ambiguity is interpreted with the received answer.

8 Use Cases

The NLQ system has been applied in various domains. Here we describe the some of the important use-cases.
Finance. We instantiated the NLQ system for the finance domain by taking data from 5 years of SEC and FDIC filings which amounts to 1.5 TB of raw data. We created an ontology combining information from two standard finance-domain ontology called FIBO (Financial Industry Business Ontology) and FRO (Financial Regulation Ontology) with the help of domain experts. We processed the data such that it adheres to our ontology and instantiated our NLQ system on top of it. The main challenge of this use case was to overcome ambiguities across various entities e.g., there are 202 different entity matches just for the token *IBM*, spreading across 6 different roles such as Company, Subsidiary, Lender, Startup, Investor, and Investee. Our interpretation engine showed great precision and recall [26] in handling such cases.
Healthcare. We also worked on a healthcare use case by building a natural language conversation service to enable the pharmacists, physicians, and nurses to easily access various drug information. In particular, information pertaining to drug classes, drug synonyms, side effects, adverse effects, symptoms, general doses is accessed from a relational database through a conversation interface. The database is described using an ontology, and the conversation service intents and entities are identified with the guidance from the ontology. The interactive nature of the conversation service allows the system to resolve ambiguities and narrow down on specific answers, e.g., if a user asks *"what is the side effect of Aspirin"* then the system can ask *"for an adult or for a child"* to give specific information.
SAP-ERP. In this use-case, we extended the NLQ technology to interact with the SAP-ERP system. Because of the large size of SAP-ERP domain (100K tables and 5M relations), this use-case posed multiple challenges regarding automatically creating the ontology, capturing the domain vocabulary and populating the translation index, as well as the query translator, which required adaptations to the SAP SQL dialect. SAP-ERP has around 63 domains and 2K sub-domains, and we have created a goal-directed algorithm to create and instantiate NLQ for each specific domain/sub-domain, leveraging meta-data information from SAP's data dictionary tables. We have instantiated our system for different domains like Sales and Distribution, FICO (FInancial accounting) and Security Administration. The details of our solution are described in [27].
Weather. We have instantiated the NLQ system on weather data such that users can ask various weather-related questions in natural language. Two important features of this use-case were 1) handling time-related expressions, and 2) domain rules related to weather and time. Consider the question - *"Will it be hot tomorrow afternoon?"* In this case, we have to map "hot" to a temperature using a domain rule (hot : temperate $\geqslant$ 40-degree celsius) and afternoon implies 12 pm to 6 pm. We also have to enable interactive disambiguation for location entities (e.g., Columbus could mean Columbus, OH or Columbus, GA).

9 Related Work

In the context of data management, natural language interfaces for relational databases (NLIDB) have been studied for several decades [4, 28]. As noted in [4, 17], early NLIDBs were based on grammars designed particularly for a specific database, thus making it difficult to incorporate other databases. The most recent work in the area is the NaLIR system [4], which operates directly on the database schema as opposed to our two-stage approach that provides physical independence, and exploits the powerful semantics of the ontology. Moreover, NaLIR mainly relies on user interaction to find the correct interpretation for ambiguous NLQs. Our system, on the other hand, provides an interpretation ranking mechanism that almost eliminates user interaction. Similar to NaLIR, Nauda [14] is an early interactive NLIDB. However, its main focus is to provide additional useful information to user, more than what is explicitly asked in the question. The PRECISE system [25] defines a subset of NLQs as semantically tractable, and generates SQL queries only for these NLQs. Similar to NaLIR, PRECISE operates directly on the database schema and thus cannot exploit the rich semantics of the ontology.

Moreover, PRECISE is not able to rank the NLQ interpretations but returns all of them to the user. We believe that interpretation ranking provides a better user experience.

Ontology-driven data access systems [7, 26, 12, 20] capture the domain semantics and provide a standard description of the domain for applications to use. Some of these works [7, 26] either focus on specific application domains or offer ontology-based data access through description logic or semantic mappings that associate queries with the underlying data stores to the elements in the ontology. However, they either fail to support multiple query types in a poly store architecture or lack a natural language interface to explore the data stored in multiple backends.

10 Conclusion

In this paper, we described our ontology-based end-to-end NLQ system to explore databases and knowledge bases. We argued the importance of using domain ontologies and entity-based reasoning in interpreting natural language queries. We illustrated our experience of instantiating the system on top of various domains. As future work, we plan to extend NLQ to support more complicated nested queries. We are also investigating how to bootstrap a conversation service using an ontology, and leverage our NLQ stack for domain specific chat bots.

References

[1] Owl 2 web ontology language document overview. `https://www.w3.org/TR/owl2-overview/`. Accessed: 2018-06-01.

[2] Snomed ct. `https://www.snomed.org/snomed-ct/what-is-snomed-ct`. Accessed: 2018-07-02.

[3] 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*. VLDB Endowment, 2002.

[4] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases–an introduction. *Natural language engineering*, 1(01):29–81, 1995.

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

[6] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *ACM SIGMOD*, 2008.

[7] D. Calvanese, G. De Giacomo, D. Lembo, et al. The mastro system for ontology-based data access. *Semant. web*, pages 43–53, 2011.

[8] B. et.al. Creation and interaction with large-scale domain-specific knowledge bases. *Proc. VLDB Endow.*, 10(12):1965–1968, Aug. 2017.

[9] R. Fernández and J. Ginzburg. Non-sentential utterances: Grammar and dialogue dynamics in corpus annotation. In *Computational linguistics*, pages 1–7. Association for Computational Linguistics, 2002.

[10] J. Ganitkevitch, B. V. Durme, and C. Callison-burch. Ppdb: The paraphrase database. In *In HLT-NAACL 2013*, 2013.

[11] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. VLDB '97, pages 436–445, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[12] E. Kharlamov, T. P. Mailis, K. Bereta, et al. A semantic approach to polystores. In *IEEE Big Data*, 2016.

[13] K. Kolhe, E. Perez, and C. Sawicki. Voice assistants (va): Competitive overview, 2018.

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

[15] H. Lee, Y. Peirsman, A. Chang, N. Chambers, M. Surdeanu, and D. Jurafsky. Stanford's multi-pass sieve coreference resolution system at the conll-2011 shared task. In *Computational Natural Language Learning*, 2011.

[16] J. Lehmann, R. Isele, M. Jakob, et al. Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web*, 2015.

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

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

[19] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *ACL (System Demonstrations)*, pages 55–60, 2014.

[20] J. McHugh, P. E. Cuddihy, J. W. Williams, et al. Integrated access to big data polystores through a knowledge-driven framework. In *IEEE Big Data, 2017*, 2017.

[21] V. Ng. Supervised noun phrase coreference research: The first fifteen years. In *Association for computational linguistics*, pages 1396–1411. Association for Computational Linguistics, 2010.

[22] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *ACM SIGMOD*, pages 39–48, 1992.

[23] H. Poon and P. Domingos. Joint unsupervised coreference resolution with markov logic. In *Empirical methods in natural language processing*, pages 650–659. Association for Computational Linguistics, 2008.

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

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

[26] 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. *Proc. VLDB Endow.*, 9(12):1209–1220, Aug. 2016.

[27] D. Saha, N. Gantayat, S. Mani, and B. Mitchell. Natural language querying in sap-erp platform. In *ESEC/FSE 2017*, pages 878–883, New York, NY, USA, 2017. ACM.

[28] 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.

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

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