Investigating PRECISE
IMPLEMENTING A PORTABLE NATURAL LANGUAGE INTERFACE TO DATABASES
NILS EVERLING
Investigating PRECISE

Implementing a portable natural language interface to databases

NILS EVERLING

Degree Project in Computer Science, DD143X
Supervisor: Michael Minock
Examiner: Örjan Ekeberg

CSC, KTH, May 2015
Abstract
A natural language interface to a database (NLIDB) lets a user query a database using a natural language. PRECISE (Popescu et al., 2003) is a formal model for a portable SQL NLIDB which interprets a question by pairing sentence tokens to database attributes and values with a maximum flow solution. PRECISE is said to be sound and complete for a large class of semantically tractable questions. We implemented PRECISE and deployed it on Geoquery, a database of geographical facts.

PRECISE made no errors in terms of returning a single, incorrect query, giving it the highest possible precision value. However, out of the 448 questions given, PRECISE was only able to produce SQL queries for 162, giving it a recall value of 0.361. A considerable amount of sentences gave rise to multiple interpretations, which prompted PRECISE to produce no query. Moreover, PRECISE by design could not produce queries for sentences which did not contain a WH-token ("what", "where", "when", "who", "which").

Our implementation of PRECISE required some manual configuration when deployed on Geoquery for best recall. While the results are tied to our implementation they give an indication of the size of the semantically tractable class as well as the portability of PRECISE.
Sammanfattning

PRECISE tolkade aldrig otvetydigt en fråga fel. Detta gav PRECISE det högsta möjliga precisionsvärdet. Däremot, utav de 448 frågor i naturligt språk som ställdes kunde PRECISE endast producera SQL-frågor för 162. Detta gav återkallsvärdet 0.361. Många frågor i naturligt tal gav multipla tolkningar vilket gjorde att PRECISE inte producerade någon SQL-fråga. PRECISE är också per design specificerad till att inte hantera frågor i naturligt tal som inte innehåller ett WH-ord ("what", "where", "when", "who", "which").

Vår implementation av PRECISE krävde en del manuell konfiguration för att få bästa återkall när den tillämpades på Geoquery. Resultaten är kopplade till vår implementation men ger en indikation av vilka frågor PRECISE klarar av, samt hur portabel PRECISE är.
1. Introduction

1.1 Purpose

A natural language interface to a database (NLIDB) provides the means for a user to query a database using a natural language. Users without knowledge of SQL and the underlying structure of a database can retrieve information by merely asking a question. An NLIDB can feel far more intuitive than other, conventional database interfaces such as search menus. Search menus usually have rigid options that might not cater to the user’s specific needs, whereas a well-tuned NLIDB can leverage the expressive power of a query language to deliver the appropriate data in a succinct manner.

As consumer computing moves from the PC to the smartphone the need for NLIDBs become more apparent. Mobile platforms have simpler interfaces that lend themselves well to NLIDBs, either by text input or speech recognition. It is important for these NLIDBs to be intelligent, reliable and effective; intelligent and reliable enough to interpret most reasonably complex sentences and effective enough to deliver results within practical time limits.

PRECISE is a formal model for a NLIDB which purportedly can handle all questions of a large class the designers of PRECISE call semantically tractable. PRECISE is said to be very portable, requiring little information of the underlying database. They have in their paper presented results of experiments where PRECISE performs favorably to other NLIDBs, as well as a theorem for soundness and completeness for semantically tractable questions[1]. This paper aims to evaluate these claims by fully implementing the model and conducting experiments equivalent those done by the designers of PRECISE[1].

1.2 Problem statement

Can PRECISE handle all semantically tractable questions? Do semantically tractable questions comprise as large of a subset of common questions as stated by the designers of PRECISE?
2. Background

2.1 NLIDB properties

The purpose of a natural language interface to a database (NLIDB) is to take a sentence in a natural language and produce a database query that corresponds to a correct semantic interpretation of the sentence. To evaluate the effectiveness of an NLIDB there are a number of metrics to consider. When testing a large number of sentences on an NLIDB (each sentence corresponding to a database query) two important metrics are recall and precision:\[2]\:

\[
\text{Recall} = \frac{\text{# of correctly produced queries}}{\text{# of sentences}} \tag{Formula 1}
\]

\[
\text{Precision} = \frac{\text{# of correctly produced queries}}{\text{# of produced queries}} \tag{Formula 2}
\]

An NLIDB might be unable to produce queries given certain sentences. This is reflected in the recall value, giving an indication of the share of sentences the NLIDB can handle correctly. Precision is also an important metric as a low value indicates the NLIDB is prone to interpret sentences incorrectly and produce wrong queries. This is highly undesirable as it will frustrate the user.

An important property to consider when evaluating an NLIDB is portability. The way the NLIDB is designed to work determines whether or not it can be reconfigured to work on different databases. NLIDBs can be classified as either domain-dependent or domain-independent:

A domain-dependent system needs particular knowledge of the underlying database in order to function. It may or may not be possible to reconfigure such a system to work on other databases. For example, an NLIDB might have been designed with a certain database in mind, having built-in logic that only makes sense in the context of that database. Such a system is called non-reconfigurable.

A reconfigurable domain-dependent NLIDB also needs knowledge of the underlying database but is designed in such a manner that this data mapping can be manually performed by a technical user. Another potential aspect of reconfiguration is the process of training. If the NLIDB works on rules of probability it draws from a pool of previous interactions to determine an answer. The NLIDB may have to be fed a large amount
of sentences and their corresponding query interpretations so it can learn and ensure precision in the future.

*Auto-reconfigurable* domain-dependent NLIDBs have an even higher degree of portability as all required knowledge is obtained in an automatic process, making it possible for non-technical users to connect the NLIDB to a database.

A *domain-independent* system does not store information about the underlying database, it relies only on a general set of rules to translate natural language sentences into database queries[3].

### 2.2 PRECISE system

In this section the PRECISE system will be introduced with an example of how it operates. First some terminology need to be introduced:

- **Element** - A database is made up of three different types of elements: relations (D_r), attributes (D_a) and values (D_v). Values are compatible with its attribute and relation, attributes are compatible with its relation.
- **Wh-value** - Words in the set {"what","which","where","who","when"}. Each database attribute has a set of compatible wh-values.
- **Token** - A token is a set of word stems that matches a database element. For example, the token \(<\text{require,experience}>\) matches the attribute element \(\text{required_experience}\). A token can potentially match several database elements, even of different types [1].

This example uses a single relation **Job** with attributes **description**, **platform** and **company**[1].

Given the question *"What are the HP jobs on a Unix system?"* the PRECISE **tokenizer** component creates a **tokenization**.

- **Tokenization** - A set of tokens whose word stems exactly make up the question, minus **syntactic markers**. Computing tokenizations is an NP-complete problem.
- **Syntactic markers** - words that have no semantic contribution to the question, such as "are" and "the"[1].

The tokenization becomes \{<what>,<HP>,<job>,<Unix>,<system>\}. The tokenizer determines what type each token is. \(<\text{what}>\), \(<\text{HP}>\) and \(<\text{Unix}>\) are **value tokens** since they match value elements. \(<\text{system}>\) is an **attribute token** and \(<\text{job}>\) is a **relation token**. All information regarding
the elements and matching tokens are retrieved from the lexicon component of PRECISE.

- **Lexicon** - A component of the PRECISE system that has two operations: 1) Retrieve the set of tokens that contain a given word stem. 2) Retrieve the set of database elements that match a given token. Tokens are generated automatically when deploying PRECISE on a database[1].

Synonym tokens also map to elements in the lexicon. For example, the token `<system>` in the question is a synonym for `<platform>` of the attribute `Job.platform`. These synonyms are generated automatically using a lexical database like WordNet[1][6].

With the tokenization set it is up to the matcher to pair the tokens with database elements. This is done by constructing an **attribute-value graph** (figure 1) which consists of attribute and value tokens, the attribute and value elements which they could potentially match as well as a source and sink. By solving the maximum flow problem on the attribute-value graph, a mapping of tokens to value elements can be derived from the paths in the flow. The graph, going in columns from left to right, has the following configuration (figure 1):

- The source has one edge to each value token.
- Value tokens have edges to value elements they match.
- Value elements have edges to their compatible attribute elements.
- The attribute element column is followed by a duplicate column consisting of the same nodes. Each attribute element in the first column has an edge to the corresponding attribute element in the second column.
- The next column consists of attribute tokens. Attribute elements in the second attribute element column have edges to matching tokens.
- The second to last column has two nodes, E and I. E stands for explicit attributes and I for implicit attributes. Every attribute token has an edge to the E node. Every attribute element in the second attribute element column has an edge to the I node.
- E and I each have an edge to the sink, T. The capacity of the edge from E to T is equal to the amount of attribute tokens in the graph. The capacity of the edge from I to T is equal to the amount of value tokens minus the amount of attribute tokens.
- All other edges in the graph have capacity = 1.
• The maximum flow of the graph must be equal to the amount of value tokens[1].

Figure 1. The attribute-value graph for the question "What are the HP jobs on a Unix system?". The blue lines represent edges chosen in the maximum flow solution. Dotted lines represent edges not chosen. PRECISE decides that <HP> must refer to company and not platform, which is the only attribute <Unix> can refer to.

PRECISE also ensures that every token-element edge abides by the attachment function:

• Attachment - Relationships between tokens derived from the parse tree of the sentence produced by PRECISE’s parser component. For example, in the question “What French restaurants are located downtown?” the tokens <located> and <downtown> are attached but <what> and <downtown> are not. Attachment is a boolean expression[1][4].

A token can only be paired to a certain element if that pairing reflects an attachment, or no attachment exists for the involved tokens. Not all tokens have attachments[4].

One of the features of PRECISE is the ability to determine implicit attributes, i.e. attributes not explicitly named in the question. The <HP> token of "What are the HP jobs on a Unix system?" could refer to a value of the company attribute as well as of the platform attribute, but neither are mentioned in reference to HP (Job is the relation). The maximum flow solution decides that <HP> must refer to company, since <Unix> must flow to the platform node.
If the tokenization contains relation tokens the matcher constructs a relation graph in a similar but simpler way:

- The source has one edge to each relation token.
- Relation tokens have one edge each matching relation element.
- Relation elements each have one edge to the sink[1].

For a complete matching the maximum flow of the relation graph must be equal to the number of relation tokens. Each relation element must also correspond to a compatible attribute or value element chosen in the attribute-value graph. In the case of the example, the only token is <job> which maps to the relation element **Job**.

After the matcher has found a mapping the equivalence checker if the problem has multiple distinct solutions that produce distinct SQL queries.

If there is only one distinct SQL query the query generator generates the SQL output of the PRECISE system. SELECT is followed by the database element paired with the WH-value, the FROM clause contains the relations used and the WHERE clause contains value constraints. If the query consists of multiple relations the generator adds join conditions in the WHERE clause. These join conditions are based on the join paths configured for the database[1].

- A **join path** is a set of equality constraints between the attributes of two or more relations[4].

PRECISE will output:

```
SELECT job.description FROM job WHERE job.company='HP' AND job.platform='Unix';
```

This concludes the example. **Figure 2** shows the overall system architecture[1].
2.3 Semantic tractability

With the PRECISE system outlined and all major components described we can now use the notion of a valid mapping:

- **Valid mapping** - Given a question q with tokenization T, lexicon L, attachment function AT and a set of database elements E, a mapping from T to E is valid if and only if:
  - Tokens in T and elements in E match one-to-one.
  - Each attribute token in q corresponds to a unique value token. Each pairing must respect the attachment function AT, that is AT(attribute token, value token) must be TRUE if the attachment function has mappings for the tokens.
  - Value tokens that do not correspond to an attribute token in q must correspond some other database attribute in E. These attribute tokens are called implicit attributes.
  - Each relation token corresponds to either an attribute or value token[1].

The class of **semantically tractable** questions are all questions q with tokenization T where:

- All tokens in T are distinct.
- T contains at least one token matching a wh-value.
- There exists a valid mapping[1].
The PRECISE designers claim that soundness (i.e., all queries produced are valid) and completeness (i.e., all valid queries can be produced) holds with regards to the semantically tractable class on PRECISE. [1]

2.4 Previous tests on PRECISE

To test PRECISE and to get an idea of how large the semantically tractable class is, the designers of PRECISE have conducted tests using their own implementation (which is not publicly available). They have determined precision and recall tests by deploying PRECISE on three publicly available databases, containing US geography facts, job listings in Texas and restaurants in northern California respectively. They have then compared the performance of PRECISE to that of a learning NLIDB developed by the curators of the databases, Lappoon R. Tang and Raymond J. Mooney [1][2]. They have also compared PRECISE to an NLIDB for Microsoft SQL Server called English Query. Using the same question set as the original experiments by Tang and Mooney, PRECISE had slightly lower recall in the case of the restaurant and geography databases, but significantly higher recall on the jobs database (figure 3).

![Figure 3](image.png)

**Figure 3.** Recall values from experiments by PRECISE designers. Recall on the geography database (Geoquery) was 0.775.
Figure 4. Precision values from experiments by PRECISE designers. PRECISE had no errors.

It is highlighted that while PRECISE has lower recall than the Mooney NLI in some cases, PRECISE is a highly portable system that does not require training[1]. In an overview of modern NLIDBs PRECISE was classified as *domain-independent*[3]. However, the designers of PRECISE acknowledge that the parser component may need to be trained to perform well on specific databases. PRECISE needs correct attachment relationships and part-of-speech tagging (POS) to function. They hold that PRECISE is sound and complete for the class of semantically tractable questions, given that the information from the parser and lexicon is correct [4].
3. Method

3.1 Testing
PRECISE is a formal model and to evaluate its practical use it had to be implemented and tested. Since the authors have not made their implementation public we have implemented our own. The main source of information for this was the 2003 paper outlining PRECISE [1]. Whenever ambiguities arose, the subsequent paper [4] was assessed. In some cases the information available still left parts of the system ambiguous so we made own interpretations. These details will be disclosed later in this section.

The novel idea behind PRECISE is the matcher component, so that component has been the focal point of our tests. To achieve this we have designed our implementation to transfer responsibility to the user in some areas outside of matcher. This way we could ensure an testing ideal environment. See Attachment and Equality Checker in section 3.2.

Our implementation of PRECISE was deployed on Geoquery, a publicly available and moderately sized database of US geography facts. Geoquery was one of the databases the PRECISE designers tested precision and recall on (figure 1 and 2). For us, using the same database makes it easy to directly compare results. We chose Geoquery since it contains several relation schemas (table 1) and those schemas can be joined in intricate ways that reflect common usage scenarios. While Geoquery is a set of prolog assertions it was interpreted as an equivalent SQL database.

<table>
<thead>
<tr>
<th>state</th>
<th>name, abbreviation, capital, population, area, state_number, city1, city2, city3, city4</th>
</tr>
</thead>
<tbody>
<tr>
<td>city</td>
<td>state, state_abbreviation, name, population</td>
</tr>
<tr>
<td>river</td>
<td>name, length, states_through_which_it_flows</td>
</tr>
<tr>
<td>border</td>
<td>state, state_abbreviation, states_that_border_it</td>
</tr>
<tr>
<td>highlow</td>
<td>state, state_abbreviation, highest_point, highest_elevation, lowest_point, lowest_elevation</td>
</tr>
<tr>
<td>mountain</td>
<td>state, state_abbreviation, name, height</td>
</tr>
<tr>
<td>road</td>
<td>number, states_it_passes_through</td>
</tr>
<tr>
<td>lake</td>
<td>name, area, states_it_is_in</td>
</tr>
</tbody>
</table>

Table 1. Relation schemas of Geoquery.
The Geoquery database comes with a set of 880 questions that were our basis for evaluation of recall and precision. The set constitutes many different types of questions and they range from very simple ("What is the capital of Utah?") to slightly trickier ("What rivers run through the states that border the state with the capital Atlanta?").

We omitted a large portion of the questions from the experiments since they required SQL aggregation functions (SUM, MAX, AVERAGE etc), something our implementation does not support. The outputted queries from PRECISE were not verified against a physical SQL database, instead they were manually evaluated only. While SQL queries can be very advanced, the ones outputted by our implementation usually only contained one or two relations, which made them quite easy to evaluate for correctness.

The PRECISE implementation was programmed in the Java language. The implementation was tested on a desktop computer equipped with a 3.2 GHz quad-core Intel i5 Ivy Bridge processor, 8 gigabytes of RAM and running Ubuntu 14.10.

3.2 PRECISE implementation

Here we disclose how we have interpreted ambiguities in the PRECISE system, as well the configuration and what limitations we have imposed on our setup.

3.2.1 Syntactic markers
The set we used consisted of:
are, the, on, a, in, is, be, of, do, with, have, has

3.2.2 WH-values
Every database attribute has a set of compatible WH-values. The only words that were used in the Geoquery question set were "what", "where" and "which". They make semantic sense for many of the attributes in Geoquery, such as state.capital, highlow.highest_point or river.states_through_which_it_flows. For our setup we let these WH-values be used interchangeably on all attributes, as situations where we needed to separate them to mitigate ambiguity did not arise.
3.2.3 Tokenization
Computing the set of tokenizations is an NP-complete problem [1]. In our implementation the computation is done by brute force, generating every single permutation of the set of matching tokens and checking each set if it passes the requirements for a tokenization.

3.2.4 Matcher
The matcher with its attribute-value graph and relation graph is the main innovation of PRECISE. There are however ambiguities that aren’t brought up by the designers and are tightly tied to the notion of a tokenization. Tokens can match elements of different types. If a token matches both a relation and an attribute, or some other combination, it is unclear what it should be used as. Every role a token has applies different constraints on either the attribute-value graph or relation graph. We chose to run the matcher on every possible permutation of a given tokenization with the constraint that a token could only be used as one type in any given scenario. This is based on the assumption that a word or phrase would never refer to multiple separate entities at the same time.

3.2.5 Attachment filtering
Attachment information is a vital part of PRECISE. It is in many cases essential to strip away improper token-element edges in the attribute-value graph. Since the process of filtering token-element edges in the attribute-value graph is not formally defined, we have had to define it ourselves. Through observation of the matcher component we have divided the rules into three different categories: WH-token to attribute element (columns 2-3, figure 1), value token to value element (columns 2-3, figure 1) and attribute token to attribute element edges (columns 5-6, figure 1). If an attachment mapping exists for at least one of the involved tokens then an edge must abide by one of the rules of the relevant categories.

**WH token $T_{WH}$ to attribute element $E_A$**
For attachment $= [T_{WH}, T_R]$ an element $E_A$ is a valid mapping with $T_{WH}$ if $E_A$ is the primary key of a relation $E_R$ which $T_R$ maps to.

The motivation for the primary key restriction is that a question like “In which state is Rochester?” with attachment [which,state] it would only make sense to return the key of the state relation, i.e state.name.
For attachment = $[T_{WH}, T_A]$ an element $E_A$ is a valid mapping with $T_{WH}$ if $T_A$ matches $E_A$.

**Value token $T_V$ to value element $E_V$**

For attachment = $[T_V, T_A]$ an element $E_V$ is a valid mapping with $T_V$ if $E_V$ is compatible with the same relation as an attribute $E_A$ which $T_A$ maps to.

Without this restriction a question like “What is the population of California?” with attachment [population, california] would yield all possible SQL queries with attribute equality constraints for “california” like $\text{mountain.state} = '\text{california}'$ or $\text{river.state} = '\text{california}'$. The only relevant attribute is $\text{state.name}$ since it’s in the same relation as $\text{state.population}$.

For attachment = $[T_V, T_R]$ an element $E_V$ is a valid mapping with $T_V$ if $E_V$ is compatible with the relation $E_R$ which $T_R$ maps to.

**Attribute token $T_A$ to attribute element $E_A$**

For attachment = $[T_A, T_R]$ an element $E_A$ is a valid mapping with $T_A$ if $T_A$ is compatible to a relation $E_R$ which $T_R$ maps to.

For attachment = $[T_A, T_V]$ an element $E_A$ is a valid mapping with $T_A$ if an element $E_V$ matching $T_V$ is compatible with $E_{AK}$, and $E_{AK}$ is the primary key for the relation $E_R$ which $E_A$ is also compatible with.

The latter condition seems very restrictive, but it is necessary to remove certain ambiguities that would arise otherwise. A question like “What is the population of California?” with attachments [what, population] and [population, california] would otherwise not yield a attribute-value graph with one sole solution (figure 5).
Figure 5. An ambiguous attribute-value graph generated by attachment filtering set too lenient. Both city.population and state.population have edges to the explicit <population> token.

While [what,population] narrows the selection of WH-paired attributes to state.population and city.population, [population,california] remains to correctly map the attribute element state.population to the attribute token <population>. Merely checking if the relation of the attribute contains a “california” value somewhere is not enough, since “california” is a value in both the state and city relation (state.name and city.state). Therefore we made the assertion that $E_{Ak}$ has to refer to a primary key.

In our implementation we use the Stanford Parser [5] to derive attachment mappings. We designed our PRECISE GUI so that we could evaluate and manually edit attachment mappings to fit the dependency model. This way we could always assert that the attachment function was correct.

3.2.6 Join paths
For our tests on Geoquery we only had one join path:

state.name = city.state = river.states_through_which_it_flows = border.states_that_border_it = highlow.state = mountain.state = road.states_it_passes_through = lake.states_it_is_in

3.2.7 Equality checker
The PRECISE system includes a component which evaluates whether there exists multiple distinct solutions to the max-flow problems in the matcher-component and if so, evaluates whether they produce semantically different SQL queries. SQL queries can look different from each other but still be equal, as table joins can sometimes be utilized in
different ways to achieve a desired result. For the sake of simplicity and better insight we skipped to implement a feature which checks SQL equality. Our implementation generates all possible queries of any given question. When testing, we manually evaluated whether multiple generated queries differed. Naturally, we interpreted the generation of multiple distinct queries from a single question as a non-answer, since PRECISE is specified to behave that way.

3.2.8 Aggregation and negation
Since very little information has been disclosed regarding how PRECISE handles aggregation ("What is the most populous city in America?") or negation ("Which states do not border Kansas?") no such functionality has been implemented. Designing this functionality on our own would increasingly make the test results tied to our own PRECISE implementation rather than the formal model. In the case of attachment filtering this was unavoidable as PRECISE depends on it. Questions requiring aggregation and negation functionality were omitted from the experiments. Over 400 questions remained in the set, which is still a reasonably large sample size.
4. Results

Figure 6. Distribution of answers by PRECISE on Geoquery.

Out of the 448 questions in the Geoquery set which did not require aggregation or negation, 162 were answered correctly by PRECISE (figure 6). The rest had either no solution produced or had multiple distinct solutions, which was interpreted as having no answer. Successfully answered questions include “What rivers are in Nevada?” and “What is the area of the state with the capital Albany?”. There were no questions for which PRECISE returned a single wrong query. All questions were processed instantly or near instantly on the desktop computer.

Figure 7. GUI of PRECISE with attribute-value graph and relation graph of the successfully answered question “What are the major cities in Missouri?”.
Recall and precision landed at \(\frac{162}{448} \approx 0.361\) (formula 1) and \(\frac{162}{162} = 1\) (formula 2) respectively. Precision as the same as previously reported. Recall was driven down by the 286 questions for which PRECISE gave no answer (figure 9). The largest share of these questions were ones containing no WH-token e.g. “How long is Rio Grande?” and “States bordering Iowa?”. The Geoquery question set contained 94 such questions.

![Recall and precision](image)

**Figure 8.** Recall and precision for PRECISE on Geoquery.

17 sentences contained foreign tokens, words which the lexicon did not recognize as part of any mapping to database elements. For the question “What major cities are located in Pennsylvania?” PRECISE could not find a token containing “located” and it was neither stripped away as a syntactic token. The lexicon did not have a mapping for the word “all” so a question such as “What are all the rivers in Texas?” could not be answered either.

![Distribution of non-answers](image)

**Figure 9.** Unanswered questions by PRECISE on Geoquery.
45 questions had words that could all be identified as tokens but had no complete tokenization where the tokens were unique (i.e. only used once) and perfectly aligned with the question. Such a question was “What states border states that border states that border Florida?”. 

41 questions had a complete tokenization but for which the matcher could not produce a sufficient attribute-value graph. The maximum flow solutions for these graphs had an insufficient flow value (not equal to the amount of value tokens) and therefore did not map all tokens according to the definition of a valid mapping.

4.1 Ambiguity

89 questions produced multiple distinct solutions. The largest contributors to this problem were the border relation (31 questions), city.population attribute (22 questions) and highlow.highest_point attribute (20 questions).

Since border had both a primary key border.state as well as foreign key border.states_that_border_it PRECISE did not sufficiently filter out incorrect interpretations. Under the set join path and attachment filtering rules PRECISE interpreted “Which states border Michigan?” (figure 10) as:

\[
\text{SELECT state.name FROM state,border WHERE border.state = 'michigan' AND state.name = border.states_that_border_it;}
\]

as well as the incorrect:

\[
\text{SELECT state.name FROM state,border WHERE border.states_that_border_it = 'michigan' AND state.name = border.states_that_border_it;}
\]

Figure 10. The ambiguous attribute-value graph for the question “Which states border Michigan?”. The graph has two maximum flow solutions, one pairing the token <michigan> to border.state and the other pairing it to border.states_that_border_it. Notice how both attribute nodes have edges to the implicit attribute node, the I-node.
PRECISE correctly interpreted most queries regarding state populations, but failed to produce unambiguous results regarding city populations. The question “What is the population of Dallas?” (figure 11) yielded the correct:

\[
\text{SELECT city.population FROM city WHERE city.name = dallas ;}
\]
as well as the incorrect:

\[
\text{SELECT city.population FROM city,state WHERE state.city2 = 'dallas' AND state.name = city.state ;}
\]

which would return all city populations in Texas.

While a question like “What is the highest point in Ohio?” might seem like it requires SQL aggregation functions, the Geoquery database contains an attribute `highlow.highest_point`. PRECISE produced a correct query for the question but also:

\[
\text{SELECT highlow.highest_point FROM highlow,city WHERE city.state = 'ohio' AND city.name = 'high point' AND city.state = highlow.state ;}
\]
mapping the stemmed token <high,point> to `city.name` (High Point, North Carolina) instead of `highlow.highest_point`. `highlow.highest_point` is still mapped to the WH-token, but as an implicit attribute. This works because WH-token to attribute element mappings are decided by attachment filtering. As "what" is compatible with all database attributes in Geoquery (see section 3.2.3) the attachment [what,high,point] filters out all attributes but `highlow.highest_point`. However, PRECISE had no problem correctly and unambiguously interpreting questions pertaining to `highlow.lowest_point`. 
5. Discussion

While our recall value is remarkably lower than the values referenced in section 2.4 it doesn’t necessarily contradict the notion that PRECISE is sound and complete for a class of semantically tractable questions. The conditions for a valid mapping (section 2.3) are restrictive, ensuring that a question is part of the semantically tractable class only if the maximum flow problem has one solution. True to its name, PRECISE had the highest possible precision value. In the cases where PRECISE interpreted questions wrong it was "saved" by the ambiguity of multiple distinct solutions, stopping it from producing a query. However, the recall value suggests that the class of semantically tractable questions with regards to a lexicon is not as big as one would hope. Most questions can in theory be semantically tractable as long as the lexicon has unambiguous token-element mappings for the words involved. The results suggest that PRECISE is less portable than previously thought. While PRECISE erroneously has been labeled as domain-independent (section 2.4) it is clear that PRECISE needs to store information about the underlying domain in the lexicon (section 2.2) in order to operate. The results show how difficult and perhaps unrealistic it is to author a single coherent lexicon where these mappings serve a large set of differently worded questions and do not interfere with each other.

Before we expound further, we must state that that our results are inherently tied to our implementation. A considerable amount of time and effort went into interpreting and implementing the PRECISE model as outlined. More so than expected, as some parts of the model specified desired system features without delving into the means of accomplishing them, such as attachment filtering.

Attachment filtering (section 3.2.5) is a most vital function to the behavior of PRECISE. Removing irrelevant token-element edges with respect to the attachment function in the attribute-value graph is in many cases necessary to get valid queries. This process was not described thoroughly, so we had to come up with a rule set of our own. These rules were based on observations of the behavior of the Matcher component. We tried to come up with a simple set of rules which would reduce ambiguity for the most common and basic questions, while not sacrificing functionality for other, perhaps more intricate questions. One of the problems in this design process was deciding how extensive our attachment filtering implementation should be. At which point are we still complying with and testing just the PRECISE model? We opted to keep our implementation reasonably simple and judge it is a reasonable approximation of the idealized filtering process in the PRECISE
papers[1][4]. It could probably be more sophisticated and yield a slightly higher recall, perhaps that could be a subject of future study.

Some aspects of PRECISE were ambiguous and forced us to make interpretations, such as the treatment of tokens with multiple types (section 3.2.4). If these interpretations are wrong with regards to the intended PRECISE model then that could have an impact on recall.

In some ways we limited the system to keep the focal point of the experiments on the matcher and lexicon components. One of these limitations regarded the equality checker component (section 3.2.7). Here we omitted the SQL equality check and instead compared SQL queries manually. This allowed us to maintain the component's idealized behavior and should therefore not have an impact on recall.

Another source of error to take into consideration is human error. Seeing as the implementation comprises roughly 3500 of lines of code it is not impossible that contains some minor bugs.

In the next section we will describe some of the difficulties we experienced tuning the lexicon component of PRECISE, and how that relates to the portability of our implementation.

## 5.1 Lexicon configuration

The lexicon configuration determines which questions are semantically tractable and which are not. The amount of unanswered questions in the categories multiple solutions, no tokenization and foreign token (figure 9) would be different given a different set of token-element mappings. Multiple solutions are results of one-to-many token-element relationships, the other two categories are results of non-existent token-element mappings.

Building the lexicon for a specific database is described as an automatic process. Database elements are tokenized and the lexical database WordNet is used to automatically generate equivalent synonym tokens (section 2.2). This lack of administrative intervention is in line with the description of an auto-reconfigurable domain-dependent NLIDB (section 2.1). In theory, PRECISE could be deployed instantly on any relational database. However, we found the automatic approach to be very erratic as our implementation would generate a lot of irrelevant synonyms. Part of speech-tagging (POS), which can help to narrow down the senses of a word, is difficult to determine automatically from database element names as they read independent of a context. Even with the correct POS-tagging a word might have several irrelevant senses which muddle the lexicon. For example, WordNet has 26 noun senses of the word
“point” in the Geoquery attribute `highlow.lowest_point`, one of which has a synonym being “state”. Hence we decided to manually adding mappings to the lexicon, rather than generating them automatically. Another reason to do this was to map relevant tokens which would not have been generated automatically otherwise. For example, to correctly answer the question “What major rivers are in Texas?” the token `<major,river>` had to be mapped to the relation `river`. The fact that all entries in the `river` relation of Geoquery denote major rivers is implicit and would not be considered in an automatic process.

One of the design challenges when authoring the lexicon was that all words in a question given to PRECISE had to be accounted for. If a word is not contained in a token or stripped away as a syntactic marker PRECISE will not continue. The questions “What rivers run through Texas?” and “What rivers do run through the state of Texas?”, though different, should preferably return equivalent SQL queries. The latter question is more verbose, containing the words “do” and “state”. One way to accommodate this variability is to map the attribute `river.states_through_which_it_flows` to every possible relevant token like `<run,through>,<do,run,through>,<do,run,through,state>` and so on. However, as more tokens are mapped to database elements the more likely it is that PRECISE finds multiple distinct tokenizations with distinct SQL queries. So while adding more token mappings is done to improve recall it might in fact have an opposite effect by creating ambiguity, as sentences can be tokenized in different ways. Increased token mappings also increase the problem size for the NP-complete problem of finding tokenizations. With enough matching tokens for a given question the tokenizer will take an unreasonable amount of time to determine the set of tokenizations. This is not something we encountered but it would be interesting to see how PRECISE performs on very large databases.

One strategy to reduce the set of needed tokens mapped to each database element is to expand the set of syntactic markers. In our set we chose to include words such as “have” and “do” in the set. This removed the need to map every relevant database attribute to a token containing them ( `<have,capital>, <do,have,name> ` etc). However, this strategy also increases the risk for ambiguity as more stripped words means less words for database elements to uniquely identify themselves with. This did not seem to be a problem on Geoquery for the limited set of syntactic markers we used but it is not unreasonable to believe that the same set might not function well on another database with other attribute naming conventions and usage scenarios.
With these practices in mind, we choose to classify our implementation of PRECISE as a reconfigurable domain-dependent NLIDB (section 2.1). This does not have to completely reflect portability of the formal PRECISE system but can be an good indication.

5.2 Future work
Further research could be made on the PRECISE model to find improvements, such as formalizing and implementing SQL aggregation and functionality. It would also be interesting to see if a more refined attachment filtering technique would yield better recall. Another premise of research would be finding databases and scenarios where the existing implementation is better suited. The formal PRECISE model could also be tweaked to be more lenient, perhaps improving recall at the cost of precision. What would happen if PRECISE did not reject a question with an unrecognized token?

Given that PRECISE could be improved to yield a reasonably high recall it would be useful to see how it performs on a very large database. If the database contains a vast amount of different values, will the NP-complete task of finding a tokenization be a problem?

6. Conclusion
PRECISE has a 0 % error rate on produced queries, giving it a precision value of 1. It can answer all semantically tractable questions, but with respect to our PRECISE implementation, that class of questions is quite small. Given the geographical database Geoquery and its associated set of questions, the semantically tractable class comprise fewer questions than would be necessary for our implementation of PRECISE to be a sufficient NLIDB. A considerable amount of sentences have multiple interpretations, which prompts PRECISE to produce no query. Recall is also driven down by sentences without WH-tokens ("what", "where", "when", "who", "which"). Our implementation of PRECISE requires some manual lexicon configuration for best results when deploying on a database.
References


Appendix

A.1 Source code

Git repository with source code is available at:
https://github.com/everling/PRECISE