CHAPTER - 4

Analysis & Design of
Natural Language Query Interface

4.1 Introduction

4.2 Architecture of Natural-English Language Interface to Database (N-ELIDB)

4.3 Algorithm for N-ELIDB

4.4 Summary
4.1 INTRODUCTION

The computer era has begun since late 1970s, bringing awareness into different working groups of society regarding the usages and benefits of computers. Not only industry, but the educational institutes, sectors like service, manufacturing, etc., also started using computers to store, process and update the information. The huge amount of data is stored in a repository called database. In order to query or retrieve information from a database by a novice, a natural language that will provide correct and precise information without knowing the depth of structured query language was the need of the time. The idea of using natural language in the context of database prompted the need for development of a system called Natural-English Language Interface to Database (N-ELIDB).

4.2 ARCHITECTURE OF NATURAL-ENGLISH LANGUAGE INTERFACE TO DATABASE (N-ELIDB)

The figure 4.1 represents the architecture of the N-ELIDB. It has two major components: (a) Linguistic Component, and (b) Database Component. The Linguistic Component translates the natural language input to an expression of Intermediate Query Representation (IQR), which is subsequently passed to Database Component for generation of Structured Query Language (SQL) statement. The resulting SQL statement is then executed by relational database management system. The Linguistic Component consists of morphological analysis, query pre-processing & context resolution, lexical analysis, syntactical analysis and semantic analysis; and Database Component consists of SQL query generation and SQL query execution.
4.2.1 Linguistic Component

The word *Linguistic* means a study of language. It has mainly three aspects to study which includes language form (syntax), language meaning and language context [46]. The computational linguistic is an interdisciplinary field known and used by computer scientists for processing of natural language. The linguistic component deals with various analyses such as morphological analysis, query pre-processing & context resolution, lexical analysis, syntactic analysis and semantic analysis.

4.2.1.1 Morphological Analysis

Morphology is the study of the way words are built up from smaller meaning-bearing units called morphemes [21]. It is also known as Lemmatization, which is a process of analyzing the token morphologically in order to find their basic forms. For example: a student has two stems of a morpheme: one is student and other is students. It takes two kinds of knowledge to correctly search for singular and plural of these forms. It allows obtaining information about words that form the sentence. For this purpose, it uses a stem
dictionary with root form of words. Table 4.1 shows some of the morpheme representation of tokens in the form of stem words.

Table 4.1: Morphemes representation of tokens

<table>
<thead>
<tr>
<th>Sample Tokens</th>
<th>Stem Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Students</td>
<td>Student</td>
</tr>
<tr>
<td>Staying, Stays, Stayed</td>
<td>Stay</td>
</tr>
<tr>
<td>Locating, Location</td>
<td>Locate</td>
</tr>
<tr>
<td>Studying, Studies</td>
<td>Study</td>
</tr>
<tr>
<td>Branches</td>
<td>Branch</td>
</tr>
<tr>
<td>Details</td>
<td>Detail</td>
</tr>
<tr>
<td>Youngest, Younger</td>
<td>Young</td>
</tr>
</tbody>
</table>

4.2.1.2 Query Pre-Processing and Context Resolution

The natural language input first undergoes a pre-processing phase in which it identifies the domain that pertains to the input query. For this, it tokenizes the input, performs morphological analyses of the words and looks them in lexicon dictionary to retrieve their syntactic and semantic properties.

The pre-processing of the input query includes: (a) stopwords removal (b) word based n-gram generation and its conversion into base words (c) spelling check (d) identifying domain, and (e) knowledge reuse.

a. Stopwords removal: Stopwords are non context bearing words, also known as noisy words which are to be excluded from the input sentence to speed up the process. The stopword removal should be done carefully; otherwise it may affect the system adversely. The list of commonly used stopwords is available in public domain [45]. The table 4.2 lists some of the stopwords used in our prototype system.
Table 4.2: List of StopWords

<table>
<thead>
<tr>
<th>StopWords</th>
</tr>
</thead>
<tbody>
<tr>
<td>again, already, amongst, any, about, against, also, anyhow, around, although, anyone, across, available, always, a, an, anything, alone, am, anyway, afterwards, along, among, another, anywhere, back, be, became, because, been, behind, beside, beyond, but, call, connect, carry, do, done, etc, either, else, everyone, everything, fill, further, here, my, nobody, nothing, other, please, he, she, somehow, it, the, thereafter, therefore, through, together, towards, whenever, your, yourself, etc.</td>
</tr>
</tbody>
</table>

b. Word based n-gram generation and its conversion into base words:
The n-gram is a contiguous sequence of n items from a given sequence of text. The items can be phonemes, letters, words or base pairs according to the application [55]. We have used this concept and convert bi-gram, tri-gram sequence into the base word as shown in table 4.4. Table 4.3 represents examples of n-gram [48].

There is another method to convert user word into base word by identifying a different form of the same word is lexicon semantic representation which is discussed in section 4.2.1.5.2. Figure 4.2 represents an algorithm to convert user word into base word.

Table 4.3 n-gram examples

<table>
<thead>
<tr>
<th>Sample Sequence</th>
<th>Unigram</th>
<th>Bi-gram</th>
<th>Tri-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>To be or not to be</td>
<td>To, be, or, not, to, be</td>
<td>To be, be or, or not, not to</td>
<td>To be or, be or not, not to be, etc.</td>
</tr>
</tbody>
</table>

Table 4.4 n-gram conversion

<table>
<thead>
<tr>
<th>Sample Sequence</th>
<th>Unigram</th>
<th>Bi-gram</th>
<th>Tri-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master of Computer Applications</td>
<td>Master, of, Computer, Applications</td>
<td>Master of, Computer Applications, of Computers</td>
<td>Master of Computer, of Computer Applications</td>
</tr>
</tbody>
</table>
c. **Spelling Check**: The detection and correction of spelling errors are an integral part of modern word-processors. In early studies, researchers have found that 80% of all the misspelled words were caused due to single character error misspelling [21] such as

- Insertion: mistyping the as ther
- Deletion: mistyping the as th
- Substitution: mistyping the as thw

One of the most popular classes of algorithm for detection and correction of spelling is minimum edit distance given by Wagner and Fisher in 1974 [21]. The minimum edit distance between two strings is the minimum number of editing operations (insertion, deletion, substitution) needed to transform one string to another. Figure 4.3 and 4.4 represent an algorithm to correct misspelled words in the query [40].

For example, the Minimum Edit Distance for two strings (a) “student” and (b) “dstudent” is 1 since one character edits operation changes it to another. (Substitute’d’ with‘s’). The possible variations are shown in table 4.5. It only substitutes single errors at a time.

---

**Figure 4.2: Algorithm to convert user word to base word**

Function ConvertBase(user query)

Input: user query UQ
Returns: BaseWord BW

[Connect to MySQL database and retrieve keyword table KT]
Get all tokens from UQ by using StringTokenizer method
For (all tokens of user query UQ) do
  Compare UQ with KT words
  If match found then
    Store KT word as BaseWord BW
  End If
End For
Return BW
Function SPCHECK (user token T1)
[Get all tokens from user Query by using StringTokenizer method T1]
Input: user tokens T1
Return: corrected tokens T11
[Read the correct spelling file CS]
Do while (CS not equal to NULL)
    ST = CALL LD (CS, T1)
    If ST equal to zero then //match found (no spell mistake)
        Break
    Else
        // no match found (spell mistake)
        Display the correct word CS in dialog box
        If user press ‘YES’ then
            correct the spelling T1 by replacing it with CS
            store it in T11
        Else
            store same user token T1 into T11
        End If
    End If
End do
Return T11

Figure 4.3: Algorithm for Spelling Detection and Correction
Function LD (CS, UT)
Input: Correct spelling CS and User Token UT
Returns: minimum value d[n][m]
M ← length(CS)
N ← length (UT)
For each column i from 0 to M do
    Store i into d[i][0]
End For
For each column j from 0 to N do
    Store j into d[0][j]
End For
For each column i from 0 to M do
    For each column j from 0 to N do
        d[i][j] = MIN(d[i - 1][j] + 1, d[i][j - 1] + 1, d[i - 1][j - 1] + c);
    End For
End For
Return d[n][m]

Function MIN (t1, t2, t3)
Returns: minimum distance MI
Store t1 as minimum MI
    If (t2 < MI) then
        Store t2 into MI
    End If
    If (t3 < MI) then
        Store t3 into MI
    End If
Return MI

Figure 4.4: Function Levenshtein Distance (LD)

Table 4.5: Variations in a token Student

<table>
<thead>
<tr>
<th>Correct Token</th>
<th>Error Token</th>
<th>Spell Check Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>UT</td>
<td></td>
</tr>
<tr>
<td>Student</td>
<td>Dstudent, Studant</td>
<td>Substitution</td>
</tr>
<tr>
<td></td>
<td>Studnt</td>
<td>Insertion</td>
</tr>
<tr>
<td></td>
<td>Studeent</td>
<td>Deletion</td>
</tr>
</tbody>
</table>
d. Identifying Domain: The degree of correct interpretation and processing of natural language queries relies on existence of the exhaustive domain specific lexicon [25]. The figure 4.5 shows domain specific lexicon for student database. The context can be resolved by identifying the domain class in form of <database name><domain name>< key terms><attribute terms> as shown in table 4.6. The algorithm for domain mapping is shown in figure 4.6.

Figure 4.5: Student Domain Lexicon

Table 4.6: Domain Identification Mapping

<table>
<thead>
<tr>
<th>Database name</th>
<th>Domain name</th>
<th>Key terms</th>
<th>Attribute terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student</td>
<td>Student</td>
<td>roll no, name, villa, residence, phone no, mobile no, department, etc.</td>
<td>stud_id, contact_no, address, birthdate, studnm, city, state, sem_no, per, branchid, gender, etc.</td>
</tr>
<tr>
<td>Exam</td>
<td>Student</td>
<td>exam, percentage, semester, branch</td>
<td>examid, examnm, sem, mks, etc.</td>
</tr>
<tr>
<td>Library</td>
<td>Student, Faculty</td>
<td>books, return, issue, library, author, edition, title, publisher, etc.</td>
<td>bookid, booknm, isbnno, authornm, publisherid, etc.</td>
</tr>
<tr>
<td>Branch</td>
<td>Student</td>
<td>department, course</td>
<td>branchid, branchnm</td>
</tr>
</tbody>
</table>
e. **Knowledge Reuse:** User can access and manipulate query which is stored in the knowledge base at any time as a template from Graphical User Interface (GUI) as shown in figure 4.7. User can select same template and without processing the linguistic analysis, can view the query result. User can also edit the existing template and compose a new query. Keeping user search history and providing a query template preserves user’s prior search effort, and gives quick starting point when he/she needs to create new queries. Figure 4.8 shows the algorithm for searching query from knowledge base and inserting query in knowledge base.
Function Search knowledge (user query UQ)

Return: flag

Store User Query UQ into HQ

[Connect to MySQL database and retrieve Knowledge base HT]

Do while (HT not null)

Compare HT query with HQ

If match found then

Set flag = 1 // search found in knowledge base

Else

Set flag = 0 // search not found in knowledge base

End If

End do

Return flag

// after successful execution of SQL query, we store the user input query into history table/log

Function Insert knowledge (User Query UQ, SQLQuery, domain name)

Return: Flag

[Connect to MySQL database and retrieve Knowledge base HI]

[Connect to MySQL database and retrieve Knowledge base HI]

// cnt is the counter to count how many times same query ask by user

// store datetime in dt
4.2.1.3 Lexical Analysis

The notion of the word is *lexeme*. The lexicon is a finite set of lexemes. The lexicon or token representation rules help in identification of lexical analysis before proceeding to syntactical analysis. The Natural Language query is divided into smaller fragments called tokens. Tokenization algorithm performs basic sentence segmentation separating strings into basic tokens. Each identified tokens can be represented by two categories: (a) using tokenization rules to identify token as value token, core token, multi-token, etc. and (b) using metadata to identify the attribute token.

4.2.1.3.1 Token representation rules: The token representation rules are used to identify token as core token, special token, numeric token, sentence ending markers, multi-token, abbreviated token, etc. Following is the partial list of token representation rules used by our prototype system:

- **Core Token:**
  - If token contains no capital letter, first letter capital, all capital letters and mixed cases, then it is a core token.

- **Numeric Token:**
  - If token contains digits only or digits separated by decimal point then it is a numeric token.
• Sentence Ending Markers:
  o Sentence can be terminated with decimal point also known as full stop (.) or question mark (?) or an exclamation (!).

• Special Value Token:
  o If token contains a sequence of capital character followed by decimal point, then it is a value token and not a terminator. For example, M.C.A. is a value token.
  o If token contains numeric data with decimal point, followed by numeric data then it is a value token and not a terminator. For example, 11234.45 is a value token.
  o If a token contains single or double quotes then it is a value token. For example, ‘mca’ or “mca” department.
  o If a token contains ‘@’ sign, then it indicates a continuous value token.
  o If a token contains apostrophe (’) followed by a character then it is treated a continuous value token. For example person’s name.
  o If a token contains ‘$’ sign or ₹ sign then it is treated as a value token. For example ₹345.

• Multi-Token:
  o If the sentence contains a word with a hyphen/dash or underscore then it is considered as a single token. For example emp_no or e-no.

• Abbreviated Token:
  o If the token contains the abbreviation then its expansion should be mapped. For example, CE Dept is mapped to Computer Engineering Department.
4.2.1.3.2 **Use of Metadata:** The metadata is data about data. The user token after going through pre-processing step gets converted into base word which is compared against metadata to decide whether it is an attribute token or not. To get metadata related to user query, first the system identifies the domain and retrieves the tables and attributes. The figure 4.9 represents an algorithm for retrieving the metadata.

![Function Metadata (domain)](image)

### Function Metadata (domain)

Return: database tables T1 and its attributes C1  
Input: Domain name  
[Connected to MySQL database ]  
Connection `con = DriverManager.getConnection(path + domain name, userName, password);`
// If connection successful,  get the metadata using `getMetaData()` method  
`DatabaseMetaData dbmd = con.getMetaData();`
// syntax of `getTables (catalog, schema pattern, table pattern, string [ ] type )`
`ResultSet rs = dbmd.getTables (null, null, null, String[ ] {"TABLE"});`
while (rs not equal to null ) do  
  Store `rs.getString ("TABLE_NAME")` into temp  
  Store `rs.getString ("COLUMN_NAME")` into col;  
  add temp into table arraylist T1  
  add col into collist C1  
End do  
Display all table arraylist T1  
Display all column names C1  
Return T1, C1

*Figure 4.9: Algorithm to retrieve metadata*

4.2.1.4 **Syntactic Analysis**

Syntactic Analysis is also called Hierarchical analysis/Parsing, is used to recognize a sentence, to allocate token groups into grammatical phrases and to assign a syntactic structure to it. The Syntactic analysis has two uses: (a) to simplify the process of subsequent component as it tries to match the meaning from the input (b) to detect the contextual meaning [21].
For example, different words like “student”, “Baroda”, “lives” which form a string as “student lives in Baroda”, but syntax makes it easy to determine who lives in Baroda by identifying subject, verb, object and its relationships. To parse a sentence, we have used Stanford Parser.

### 4.2.1.4.1 Stanford Parser

The Stanford Parser, as discussed in chapter 3, is a probabilistic parser which uses the knowledge of the language gained from hand-parsed sentence and then tries to produce the most likely analysis of new sentences. The Stanford dependencies provide a representation of grammatical relations between words in a sentence. We have developed a multi-liaison algorithm using Stanford Parser [6].

### 4.2.1.4.2 The Multi-Liaison Algorithm

The English sentences can have multiple subjects and objects. The Multi-Liaison Algorithm would display them with connecting verbs or predicates. In this algorithm, the sentence is parsed with the help of the Stanford parser and output of the parser is used as an input to our algorithm for finding all the subjects, objects and the predicates. The output of multi-liaison is shown in figure 4.18. The subjects can be either nouns or even pronouns. Besides, one subject can be related to multiple objects and vice-versa.

The algorithm is written in JAVA using Net Beans IDE 6.5 and was tested with a variety of input. This algorithm is useful for the user who follows input in the form of template <subject> <verb> <object>. It is also useful for researchers who are doing research in NLP and Text Mining [6]. The limitation of this motivated us to develop a prototype system which has no template restriction; and user can freely input the sentence in its own form.
The Detailed Algorithm

The Multi-Liaison algorithm as shown in Fig. 4.11, calls the two functions (a) Convert_sentence in which the sentence is parsed, using Stanford Parser that returns POS tagging, parse tree and typed dependency; (b) Multi-Liaison which, in turn, takes input as: the POS of each word as shown in figure 4.10 [8], the parse tree as shown in figure 4.19 and the typed dependencies as shown in figure 4.17. Two functions are then called: (a) GET_TRIPLETS (figure 4.12) and (b) GET_RELATIONSHIP (figure 4.15) respectively. Finally the multiple liaisons are displayed as shown in figure 4.18.
Chapter 4: Analysis & Design of Natural Language Query Interface

Figure 4.10: POS tag sets

| CC  | Coordinating conj.       | TO   | infinitive to |
| CD  | Cardinal number          | UH   | Interjection  |
| DT  | Determiner               | VB   | Verb, base form |
| EX  | Existential there        | VBD  | Verb, past tense |
| FW  | Foreign word             |VBG  | Verb, gerund/present part |
| IN  | Preposition              | VBN  | Verb, past participle |
| JJ  | Adjective                | VBP  | Verb, non-3rd ps. sg. present |
| JJR | Adjective, comparative   |VBZ  | Verb, 3rd ps. sg. present |
| JJL | Adjective, superlative   | WDT  | Wh-determiner |
| LS  | List item marker         | WP   | Wh-pronoun |
| MD  | Modal                    | WP$  | Possessive wh-pronoun |
| NN  | Noun, singular or mass    |WRB  | Wh-adverb |
| NNS | Noun, plural             |#    | Pound sign |
| NNP | Proper noun, singular    |$    | Dollar sign |
| NNPS| Proper noun, plural      | .    | Sentence-final punctuation |
| PDT | Predeterminer            |      | Comma |
| POS | Possessive ending        |      | Colon, semi-colon |
| PRP | Personal pronoun         | (    | Left bracket character |
| PPS | Possessive pronoun       | )    | Right bracket character |
| RB  | Adverb                   |"    | Straight double quote |
| RBR | Adverb, comparative      |    | Left open single quote |
| RBS | Adverb, superlative      | .   | Left open double quote |
| RP  | Particle                 |    | Right close single quote |
| SVM | Symbol                   | "   | Right close double quote |

Function: CONVERT_SENTENCE (Input_Str)

Returns: Output_Str

Input_Str: Sentence to be parsed

[Run the Stanford parser with Input_Str as input]

Output_Str ←

i) POS of each word

ii) The parse tree generated

iii) The typed dependencies

Function: MULTI_LIAISON (Input_Str)

Returns: Multiple liaisons or error message

Input_Str: Output_Str

T = Call Function GET_TRIPLET (Output_Str)

R = Call Function GET_RELATIONSHIP (Output_Str)

Return multiple liaisons T and R

Figure 4.11: General Multi-liaison Algorithm
Function: GET_TRIPLETS (Output_Str)
Returns: Multiple subjects, objects and predicates
[Read level 1 of Parse Tree – refer Fig. 4.19]
If tree contains ‘NP’ or ‘NNP’ then
   S= call Function GET_SUBJECT (NP sub tree)
Else
   Return error message
End If
If tree contains ‘VP’ then
   P= call Function GET_PREDICATE (VP sub tree)
   O= call Function GET_OBJECT (VP sub tree)
Else
   Return error message
End If
Return multiple S, O, P

Figure 4.12 Function GET-TRIPLET

Function: GET_SUBJECT (NP sub tree)
Returns: Subject(s) and adjective(s)
For (all nodes of NP sub tree) do
   If NP sub tree contains ‘NN’ or ‘NNP’ or ‘NNS’ then
      Store POS as a subject
   End If
   If NP sub tree contains ‘JJ’ then
      Store POS as an adjective
   End If
End For
Return the subject(s) and adjective(s)

Figure 4.13: Function GET-SUBJECT
Function: GET_PREDICATE (VP sub tree)

Returns: Predicate(s)

For (all nodes of VP sub tree) do
    If VP sub tree contains 'VB?' then
        Store POS as a predicate
    Else
        Return error message
    End If
End For

Return the predicate(s)

Figure 4.14: Function GET-PREDICATE
Function: GET_RELATIONSHIP (Output_Str)
Returns: Multiple liaisons / relations
[Read the Stanford typed dependencies from Output_Str]
For (all terms in typed dependencies) do
  If typed dependencies contain ‘NSUBJ’ then
    Store both words of NSUBJ as S1 and S2
  End If
  For each value of subject from GET_SUBJECT do
    If subject matches S2 then
      [Compare for predicates]
    End If
    For each value of predicate from GET_PREDICATE do
      If predicate matches S1 then
        [Concatenate subject and predicate as R1]
        Store R1 in the relation
      End If
    End If
    If typed dependencies contain ‘DOBJ’ or ‘PREP’ then
      Store both the words as D1 and D2
    End If
  End For
  For each value of object in GET_OBJECT do
    If object matches D2 then
      Store value of object in the relation as R2
    End If
  End For
End For
Return R1+R2
As shown in Fig. 4.12, the GET_TRIPLETS function takes the Stanford Parse Tree as an input and by considering the nodes under the NP sub tree and the VP sub tree, finds all the subjects, objects and predicates. The GET_RELATIONSHIP (figure 4.15) finds and displays the relationships between the subjects and objects using functions GET_SUBJECT (Figure 4.13), GET_PREDICATE (figure 4.14) and GET_OBJECT (figure 4.16).
Example 1: The Puja and Reema work together for Dissertation work in their college

The Stanford Parser output:
Tagging:
The/DT Puja/NNP and/CC Reema/NNP work/VBP together/RB for/IN Dissertation/JJ work/NN in/IN their/PRP$ college/NN

Parse Tree:
(ROOT
  ($
    (NP (DT The) (NNP Puja)
    (CC and)
    (NNP Reema))
    (VP (VBP work)
    (ADVP (RB together))
    (PP (IN for)
      (NP
        (NP (JJ Dissertation) (NN work))
        (PP (IN in)
          (NP (PRP$ their) (NN college))))))))

Typed Dependencies:
det(Puja-2, The-1)
nsubj(work-5, Puja-2)
conj_and(Puja-2, Reema-4)
nsubj(work-5, Reema-4)
advmod(work-5, together-6)
amod(work-9, Dissertation-8)
prep_for(work-5, work-9)
poss(college-12, their-11)
prep_in(work-9, college-12)
Chapter 4: Analysis & Design of Natural Language Query Interface

The Multi-Liaison Output:
Subject: 2
NNP Puja DT The
NNP Reema
Predicate: 1
VBP work
Object: 2
NN work JJ Dissertation
NN college PRP$ their

Relationship:
Puja - work - Dissertation - work - college
Reema - work - Dissertation - work - college

Figure 4.18: Output of Multi-Liaison Algorithm

Figure 4.19: Parse Tree for a given sentence
4.2.1.5 Semantic Analysis

Syntax driven semantic analysis can be thought as the meaning of the sentence composed by the meaning of its parts [21]. Here, the input is passed through a parser to derive its syntactic structure. The syntactic structure is, then, passed to a semantic analyzer to determine its contextual meaning. For example, syntactic constitution means sentence -> noun | verb and the semantic attachment leads to the semantic property of noun (proper noun, common noun, etc.) or a verb. The output of the semantic interpretation module gives a logical expression of the words in the lexicon, to generate the logical query. The approach used here is an Intermediate Query (IQ) representation which can express the meaning of user input in terms of high level concepts, independent of database structure. For example, if the users input in the natural language query as:

“Display all students who stay in Baroda city”, which can be translated into logical form such as:

\[
\text{acomp(give-1, stud-2)} \\
\text{dep(location-4, who-3)} \\
\text{dep(stud-2, location-4)} \\
\text{nn(city-7, Vadodara-6)} \\
\text{prep_in(location-4, city-7)}
\]

SQL Query: `select * from stud where (city like '%vadodara%').`

The logical query as stated above represents the meaning of user’s question to find the different pairs like [city, Vadodara] such that a city is a City and Vadodara is a name of a city. The logical query generated by the parsing and semantic interpretation module, expresses the meaning of the user’s question in terms of a logical concept. The logic query does not refer directly to the database object.
such as tables or columns; it does not specify how to search the database to retrieve the necessary information. In order to retrieve the information requested by the user, the logic query has to be transformed into query expressed in some structured query language supported by the underlying database management system.

We have used WordNet to identify the synonymy, hypernymy, hyponymy, etc. in the development of prototype system. The tokens which WordNet cannot identify are defined using another two methods (a) Lexicon Semantic representation and (b) Rules for identification of person name, location and date.

### 4.2.1.5.1 WordNet

We have used WordNet, as described in chapter 3, to know the semantics of particular token. WordNet [31] is an electronic lexical database of the English language developed at Princeton University under the direction of George A. Miller.

We have used Java WordNet Library (JWNL), which is an API for accessing WordNet. The major methods used from JWNL.JAR package[46] are: (i) initialize( ) for library initialization in the program (ii) Dictionary.getInstance( ) to get a current installed dictionary (iii) getIndexWord( ) is used to get the part of speech and senses (iv) PointerUtils( ) is used to find the chains of pointer of a given type.

For example, the user asks ‘give detail of student staying in Gujarat’. Here, the word ‘gujarat’ is not defined by user whether it is a person name or location. The WordNet recognizes the word and identifies it as a location. Figure 4.20 shows the algorithm for finding synonyms, hyponymy, hypernymy of a token using WordNet API [46]. The output is shown in figure 4.21.
Function: CheckWord (token)
Input Str: token
Output: Returns integer value for person, location, both or none

//initialize(JWNL\jwnl14-rc2\config\file_properties.xml);
// get constructor Dictionary.getInstance();

word = Dictionary.getIndexWord(POS,NOUN, token)
sense ← word.getsenses()
P ← length(senses)

For each column i from 0 to P do
    Ses ← sense[i]
    Sword ← ses.getwords()
    Tno ← 0
    Q ← length(sword)
    For each row j from 0 to Q do
        Hyperny ← sword.getLemma()
        Compare hyperny index with token index
        If equal then
            flag = 1
            tno ← tno + 1
            k ← ses.length
        End If
    End For
End For

If flag > 0 then
    PP ← Pointer word.getSense(TT).getPointers (PointerType. HYPONYM)
    If length(PP) is zero, flag2 = 0 else
        For each column i from 0 to tno do
            S ← ses[k]
            P ← Store Pointer S. getPointers(PointerType.HYPONYM)
            If P is zero, flag = 0
            s = s + 1
        End if
    End For
End For
End For
End if
End if

Print hyperny
Print Sense no
For (every hyperny) do
  Compare hyperny with keyword ‘person’ and ‘location’
  If match found
    Set flag = 2 for person
    Set flag = 3 for location
  End if
End for
Return flag

Figure 4.20: Algorithm for finding the person or location using the WordNet Java API.

Example 1: List of students staying in gujarat
Input: Token = Gujarat

The WordNet output:

Senses of the word 'gujarat':
First char is capital : 1,  Hyponym : 1,  Sense no : 1
hypernymy in wordnet : entity,  physical_entity
hypernymy in wordnet : object,  physical_object
hypernymy in wordnet : location
value of t in wordnet == 3

Figure 4.21: Output of Algorithm using WordNet java API
4.2.1.5.2 **Lexicon Semantic Representation:** It is another form of representation for user tokens and user input symbols in the form of semantic word as shown in table 4.7 and 4.8.

<table>
<thead>
<tr>
<th>User tokens</th>
<th>Semantic Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>phone number, mobile number, contact number, cell number,</td>
<td>contact-no</td>
</tr>
<tr>
<td>telephone no, residential number,</td>
<td></td>
</tr>
<tr>
<td>office phone no, college phone no, dial no</td>
<td></td>
</tr>
<tr>
<td>percentage above 70</td>
<td>distinction</td>
</tr>
<tr>
<td>address, place, stay, living, situated, residence, dwelling, venue, locus</td>
<td></td>
</tr>
<tr>
<td>point, domicile</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4.8: Example of Semantic representation 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>User tokens</td>
</tr>
<tr>
<td>greater than or equal to, &gt;=</td>
</tr>
<tr>
<td>less than, lower, lesser, younger, smaller, minimum, under, lowest, &lt;</td>
</tr>
<tr>
<td>less than or equal to, &lt;=</td>
</tr>
<tr>
<td>greater, greater than, not less than, older, above, larger, &gt;</td>
</tr>
<tr>
<td>equal, like, equivalent, same, similar, =</td>
</tr>
<tr>
<td>not equal to, except, other than, &lt;&gt;, !=</td>
</tr>
</tbody>
</table>

4.2.1.5.3: **Rules for Identification of Person name, Location and Date:**

The Stanford Parser and WordNet do not cover all the possible contextual words; we have used another pattern matching approach for identifying the proper nouns using various heuristic rules. The determination of proper noun helps in identifying the subject, object and to make powerful semantic lexicon.
Rules for identifying Person’s Name [87]:

- Contextual words belong to set {men, books, author, co-author, state, city, country, university, college, school, island, hero, hospital, establish, saints, chairman, director, etc.} indicate kind of proper noun.
- If set of capitalized word include a set of letters followed by (.) or followed by mostly one (rarely two) capitalized words, then the whole set is likely to be name.
- If one of the capitalized words appears subsequently, the probability for it to be the name increases.
- If the immediately preceding word to potential name is any of these words belongs to set {Mr, Shri, Prof, Dr, Mrs, Army, Airforce, Navy rank, Justice, H.H, Master, St, Ratna, Padmashri, Sir, His Excellence, Rev, Lord, Swami, Brother, Sister, etc.}, the potential name is certainly a name.
- If the preposition belongs to set {friend of, colleagues of, co-author, with, men, persons, emperor, men like, sage, as, etc.}, the probability for it to be named increases.
- If the word immediately after the capitalized word(s) (i.e., the post-position) belongs to set {said, told, etc.} the probability for it to be named increases.
- An apostrophe’s (‘s) to a capitalized word, the probability for it to be named increases.

Rules for identifying Place/Institute/Organization Name

- If a Preposition ‘of’ comes immediately after a name, it is likely to be a place or organization or institute.
- Possible set of preposition for potential Place or Organisation is {from, in, at, to, for, of, etc.}
- If the set of words or one of capitalized words appears at the beginning of a sentence, it is more likely to be named.
4.2.2 Database Component

The database component consists of SQL Generation and SQL Execution. The SQL Generation component takes an Intermediate Query (IQ) as an input from Linguistic component and generates an equivalent SQL query as output. The SQL Execution component executes the generated SQL query and displays appropriate query output or message on GUI to user.

4.2.2.1 SQL Query Generation

This component takes input from Linguistic component in the form of intermediate query representation and generates SQL queries. Table 4.9 gives the representation of SQL grammar for some of the Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query/Retrieval Language (DQL) statements [58].

The steps to generate SQL query are: (i) to analyze the user words from an input sentence (ii) to recognize the specific patterns of database query language (iii) to determine the analysis of answer and (iv) by using an IQ, the system generates structured query. This process is discussed in algorithm of SQL generation (section 4.2.2.1.2) and output is shown in the figure 4.22. The precise output with all formats of SELECT statement is discussed in chapter 5. The algorithms which do not require the intermediate representation are also shown for DDL and DML statements in figure 4.23, 4.24 and 4.25. The outputs of DDL and DML statements are discussed in chapter 5.

Rules for identifying Date

- Words belong to set \{century, decade, AD., BC. during, before, after, until, since, etc.;\} then probably it is a year.
Table 4.9: SQL Grammar

### SQL Grammar for SELECT Statement

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;select query&gt;</code></td>
<td>:= <code>&lt;select&gt; [ALL] [DISTINCT] &lt;select query&gt;</code></td>
</tr>
<tr>
<td><code>&lt;select&gt;</code></td>
<td>:= SELECT <code>&lt;select list&gt;</code> <code>&lt;from clause&gt;</code> [ <code>&lt;where clause&gt;</code> ]</td>
</tr>
<tr>
<td><code>&lt;select list&gt;</code></td>
<td>:= <code>'*'</code></td>
</tr>
<tr>
<td><code>&lt;column element&gt;</code></td>
<td>:= <code>&lt;column&gt;</code></td>
</tr>
<tr>
<td><code>&lt;from clause&gt;</code></td>
<td>:= FROM <code>&lt;table reference&gt;</code>[<code>${',' </code>&lt;table reference&gt;<code>}</code>...]</td>
</tr>
<tr>
<td><code>&lt;table name&gt;</code></td>
<td>:= <code>tablename</code></td>
</tr>
<tr>
<td><code>&lt;where clause&gt;</code></td>
<td>:= WHERE <code>&lt;search conditions&gt;</code></td>
</tr>
<tr>
<td><code>&lt;search condition&gt;</code></td>
<td>:= <code>&lt;condition&gt;</code> with logical predicate (AND/OR/NOT)</td>
</tr>
<tr>
<td><code>&lt;condition&gt;</code></td>
<td>:= Expression in form of $X_R^Y$</td>
</tr>
<tr>
<td><code>&lt;group by&gt;</code></td>
<td>:= <code>&lt;column element&gt;</code></td>
</tr>
<tr>
<td><code>&lt;order by&gt;</code></td>
<td>:= <code>&lt;column element&gt;</code> [<code>desc</code>]</td>
</tr>
<tr>
<td>Note: X and Y are the set of values representing column, constant and R can be operator like <code>{&lt;,&gt;,&lt;=, &gt;=, !=,==}</code></td>
<td></td>
</tr>
</tbody>
</table>

### SQL Grammar for DML Statements (like Insert, Update, Delete)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Insert Query&gt;</code></td>
<td>:= <code>&lt;insert&gt;</code> INTO <code>tablename</code> VALUES <code>&lt;values&gt;</code></td>
</tr>
<tr>
<td><code>&lt;insert&gt;</code></td>
<td>:= INSERT <code>[&lt;column element&gt;]</code> <code>tablename</code></td>
</tr>
<tr>
<td><code>&lt;column element&gt;</code></td>
<td>:= <code>&lt;column&gt;</code></td>
</tr>
<tr>
<td><code>&lt;values&gt;</code></td>
<td>:= set of values representing columns or constant value</td>
</tr>
<tr>
<td><code>&lt;Update Query&gt;</code></td>
<td>:= <code>&lt;Update&gt;</code> SET <code>tablename</code> <code>&lt;expression&gt;</code> = <code>&lt;expression&gt;</code> [WHERE <code>&lt;condition&gt;</code>]</td>
</tr>
<tr>
<td><code>&lt;Update&gt;</code></td>
<td>:= UPDATE <code>tablename</code> <code>column</code> <code>&lt;expression&gt;</code> = <code>&lt;expression&gt;</code></td>
</tr>
<tr>
<td><code>&lt;condition&gt;</code></td>
<td>:= Expression in form of $X_R^Y$</td>
</tr>
<tr>
<td>Note: X is the set of values representing column, Y can be value constant and R can be operator like <code>{+,*,-, /, %}</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Delete Query&gt;</code></td>
<td>:= <code>delete</code> FROM <code>tablename</code> [WHERE <code>&lt;condition&gt;</code>]</td>
</tr>
<tr>
<td><code>&lt;delete&gt;</code></td>
<td>:= DELETE <code>tablename</code></td>
</tr>
<tr>
<td><code>&lt;condition&gt;</code></td>
<td>:= Expression in form of $X_R^Y$</td>
</tr>
<tr>
<td>Note: X and Y are the set of values representing column, constant and R can be operator like <code>{&lt;,&gt;,&lt;=, &gt;=, !=,==}</code></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 4: Analysis & Design of Natural Language Query Interface

### SQL Grammar for DDL Statements (like Create, Drop)

<table>
<thead>
<tr>
<th>SQL Grammar</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;Create Statement&gt;</code> := <code>&lt;create statement&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;table reference&gt;</code> := <code>(&lt;column element&gt;)</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;datatype&gt;</code> := <code>int/char/varchar2/float/date</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;constraint&gt;</code> := <code>primary key/reference key/unique key/not null/check</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;Create statement&gt;</code> := <code>CREATE TABLE</code> tablename</td>
<td></td>
</tr>
<tr>
<td><code>&lt;column element&gt;</code> := <code>column</code></td>
<td></td>
</tr>
<tr>
<td><code>Drop statement</code> := <code>DROP TABLE</code> tablename</td>
<td></td>
</tr>
</tbody>
</table>

4.2.2.1.1 **Answer Analysis:** The question class of the user input query is received from linguistic analysis, which gives output in the form of intermediate representations, wherein each token or lexeme is classified according to answer class as shown in table 4.10, and then it generates the SQL query.

<table>
<thead>
<tr>
<th>Question class</th>
<th>Answer class</th>
<th>Example of Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>How</td>
<td>count</td>
<td>how many students in semester 4?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>how many students whose city is godhra?</td>
</tr>
<tr>
<td>Who</td>
<td>person</td>
<td>who is student misha?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list of students who study in semester 3</td>
</tr>
<tr>
<td>What</td>
<td>attribute</td>
<td>what is the name of student who lives in city baroda</td>
</tr>
<tr>
<td>Where</td>
<td>place</td>
<td>where does student anup live?</td>
</tr>
<tr>
<td>Word ending with ‘wise’</td>
<td>order by</td>
<td>branchwise list of students</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list of students Genderwise</td>
</tr>
<tr>
<td>Word ending with ‘wise’</td>
<td>group by</td>
<td>count total number of students genderwise</td>
</tr>
<tr>
<td></td>
<td></td>
<td>total number of students citywise</td>
</tr>
<tr>
<td>All</td>
<td>‘*’</td>
<td>list all students</td>
</tr>
<tr>
<td>Gender</td>
<td>male or female</td>
<td>list all female students</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list of students whose gender is male</td>
</tr>
<tr>
<td>Location</td>
<td>city</td>
<td>list of students who live in baroda</td>
</tr>
<tr>
<td></td>
<td>state</td>
<td>list of students from gujarat state?</td>
</tr>
<tr>
<td>Score more</td>
<td>percentage</td>
<td>list of students who score more than 50 %</td>
</tr>
<tr>
<td>Score less</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.10: Answer Analysis
<table>
<thead>
<tr>
<th>Born</th>
<th>birthdate</th>
<th>list of students who born in month of april</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>creation of table</td>
<td>make the table emp with fields empno as int</td>
</tr>
<tr>
<td>Insert</td>
<td>insertion of record into table</td>
<td>add the data into table emp with value 11</td>
</tr>
<tr>
<td>Update</td>
<td>updation of value into table</td>
<td>change the data into table emp with value empno=12</td>
</tr>
<tr>
<td>Delete</td>
<td>deletion of record from table</td>
<td>remove the data from table emp</td>
</tr>
<tr>
<td>Drop</td>
<td>delete table structure</td>
<td>cancel the table emp</td>
</tr>
</tbody>
</table>

### 4.2.2.1.2 Algorithm of SQL Generation from IQ representation:

Step 1: Do while (all intermediate representation IQ) not null

- Compare for every dependency representation, AR (G, D) (section 3.1.2.1)
  
  // challenge is to determine when two attribute occurs in a user query, one might be attribute or stop word and other might be condition.

Step 2: // determine attribute of select clause

- If G or D matches with metadata() then
  
  Store G or D into attribute ATT

- Else If G or D matches with word ‘all’ then
  
  Store ‘*’ into attribute ATT

- Else If G or D matches with the word ‘how’ and ‘many’ then
  
  Store count(*) into attribute ATT

- Else If G or D matches with the word ‘unique’ then
  
  Store unique (G) or unique (D) into attribute ATT

- Else

  Store ‘*’ as default attribute ATT

End If
Step 3: // determine condition of SQL query for SELECT statements. The condition can be single or multiple conditions. Moreover, if order by clause is executed, then it should be last statement according to SQL grammar (table 4.10).

If G or D matches with word ‘order’ or ending with ‘wise’ then

    If the word ‘count’ not specified then
        Store order by ATT in Condition CON
    End If

Else If G or D matches in word ‘group’ or ending with ‘wise’ then

    If the word ‘count’ is specified then
        Store group by ATT in condition CON
    End If

Else If G or D matches with word ‘city’, ‘state’ then

    // Compare whether a city or state is specified as an attribute or condition
    Compare for DR (G=’city’ or D=’city’) or DR (G=’state’ or D=’state’)
    If it matches then
        Store where condition of city or state into CON
    End If

Else if G or D matches with word ‘birthdate’ then

    Compare for keyword ‘month’,
    If ‘month’ or ‘year’ or ‘day’ is asked then
        Store where condition of birthday into CON
    Else If ‘today’ keyword then
        Store where condition of birthday into CON
    Else If ‘tomorrow’ keyword then
        Store where condition of birthday + 1 into CON
    Else If ‘yesterday’ keyword then
        Store where condition of birthday-1 into CON
    End If

Else if G or D matches with word ‘percentage’ then

    //Compare for digit pattern
If percentage is in digit then
    Compare for dependency,
    Store ‘where condition of percentage’ into CON
End If
Else if G or D matches with word ‘like’ then
    // Compare for pattern matching
    Store ‘where condition of pattern’ into CON
If G or D matches with word ‘younger’ then
    Store ‘where condition of birthdate’ with maximum function
Else if G or D matches with word ‘older’ then
    Store ‘where condition of birthdate’ with minimum function
End If
If G or D does not match with any condition above then
    Make use of WordNet() to find the person or location and
    Store ‘where condition of place or person’ into CON
End If

Step 4: Generate SQL query by combining Entity, Attribute ATT and Condition CON
using SQL grammar BNF format (table 4.10).

Note: All conditions are generated for SELECT statement taking student domain. The
various categories of condition and its output are given in chapter 5.
Example 1 : list name, address and phone no of student who live in Surat city

**DOMAIN** : stud **DATABASE NAME** : stud

Base str= give studnm, address and contact_no of stud who location in Surat city

**POS tagging** : words: give/VB studnm/NN address/NN and/CC contact_no/NN of/IN stud/NN who/ WP location/VBN in/IN Surat/NNP city/NN

Intermediate representation:

- iobj(give-1, studnm-2)
- dobj(give-1, address-3)
- nsubj(location-9, address-3)
- dobj(give-1, contact_no-5)
- conj_and(address-3, contact_no-5)
- prep_of(address-3, stud-7)
- rcmid(address-3, location-9)
- nn(city-12, Surat-11)
- prep_in(location-9, city-12)

**Entity** : stud;

**Attribute** : studnm; address; contact_no;

**Filtering condition** : (city like '%surat%');

**Generated SQL** = select studnm, address, contact_no from stud where (city like '%surat%');

*Figure 4.22: Output of IQ to SQL Generation (SELECT statement)*
Function Insert (User Query UQ)
Returns: Generated Insert Statement CT

Input: User Query // add the data into table t1 with value 11, ami, 89000
Tokenize the UQ with keyword ‘table’ and ‘value’ as delimiter and store it into TB and VB
// generate Insert statement
CT= Insert into TB values VB
Return CT

Function Update (User Query UQ)
Returns: Generated Update Statement CT

Input: User Query UQ // edit the table t1 with value sal = sal *0.5 where empno =1
Tokenize the UQ with keyword “table”, “value” and “where” clause and store it into TB, VB and WH.
// Generate Update statement
CT= Update TB set VB=expression where WH
Return CT

Function Delete (User Query UQ)
Returns: Generated Delete Statement CT

Input: User Query UQ // delete all the data from emp table or delete the data from emp table where id = 1
Tokenize the UQ with keyword “table”, “where” and store into TB and WH.
// Generate Delete statement
If “where” clause not present then
CT= Delete from TB
Else
CT= Delete from TB where WH.
End If
Return CT

Figure 4.23: Algorithm of IQ to SQL Generation (DML statement)
Figure 4.24: Algorithm of IQ to SQL Generation (DDL statement)

Function Createtable (User Query UQ)
Returns: Generated Create Statement CT
Input: User Query // make table t1 with field empno as int, empnm as varchar, sal as float
Tokenize the UQ with keyword ‘table’, ‘field’ and ‘as’ as delimiter and store it into TB, FT, DT
[ Change the datatype of user specified with MySQL datatype]
// Generate SQL statement
CT= Create table TB( FT DT, FT DT)
Return CT

Function Drop (User Query UQ)
Returns: Generated Drop Statement CT
Input: User Query // remove table emp
Tokenize the UQ with keyword “table “and store it as TB
// generate Drop statement
CT= Drop table TB
Return CT

Function Retrieve (User Query UQ)
Returns: CT
Input : User Query UQ // select from table emp
Tokenize the UQ with keyword “table” and store it into TB
//Generate Select statement
CT= Select * from TB
Return CT

Figure 4.25: Algorithm of IQ to SQL Generation (retrieve method)
4.2.2.2 SQL Query Executor

The task of query executor is to map the generated SQL query to database adaptor as discussed in section 4.3.2.1 and retrieve the relevant answer by connecting to the appropriate database tool. We have used a MySQL relational database for query execution. The SQL query which is executed by relational database giving following kind of responses:

- The retrieved database tuples contain answers and are stored in file rather than displaying directly in grid format as specified by the SQL database output.
- When the system may not able to search the particular tuples, then the appropriate response is sent to the user. For example, no record found.
- When the system may not understand the question, then no tuples will be retrieved and system may send appropriate response to the user. For example, the query cannot be generated or no domain found.
- User can select template from system knowledge base and without processing the linguistic analysis, can view the query result. User can also edit the existing template and compose a new query.

4.2.2.2.1 Database Adaptor: The structured data is stored in the form of a flat file or relational database [5]. A flat file is a plain text or mixed text usually containing one record per line. The data arrangement consists of a series of rows and columns organized in a tabular format [5]. The record does not have a relationship with each other. Within a record, single field can be delimited by any delimiter or some fixed length as a field separator. We have developed a data convertor tool Any2MySQL that transforms structured data (plain text files or database files) into relational database - MySQL. Figure
4.26 shows the general architecture of the conversion of any data to the MySQL relational database system. The figure 4.27, 4.28 display the output of general algorithm for data conversion tool. The figure 4.29 shows the conversion of excel file into MySQL database.

**Figure 4.26: Architecture of the Any2MySQL system**

### 4.2.2.2.1.1: General Algorithm for Data Conversion

Step 1: Select text file (.csv or .txt file), Excel file (.xls file), MS-Access file (.mdb file), Oracle file, SQL Server file, Extensible Markup Language file (.xml) as an input to the system through Interface.

Step 2: For Oracle selected file, Input Server name, port number, sid, username and password through Interface.

Step 3: Create a connection link to selected source.

Step 4: After connection – Get Metadata- table name, attribute names, data types through system.

Step 5: Create connection link - MySQL database.

Step 6: Compare the data types of source and destination files.

Step 7: Formulate a CREATE query of each table and Execute.

Step 8: Formulate an Insert query of each record for each table and Execute.

Step 9: Refresh query browser of MySQL and view the tables and records.
4.2.2.1.2: Rules for Data Convertor

Rules while retrieving data from Flat File

- When reading a sentence, using comma as a delimiter, make sure that fields with embedded commas must be quoted. For example: 101, “Ami patel, nana bazar”.
- Input File should have extension .txt or .csv

Rules while retrieving data from Spreadsheet

- No need to create Data Source Name (DSN) from control panel.
- Input file should have .xls extension
- Data types “Number” and “Date/Time” from the spreadsheet is converted to Numeric and Datetime respectively in MySQL.
- For each sheet of .xls file, a separate table is created having the same name as that of sheet. The field name in a table will have same column name in respective excel sheet. It gives option of converting individual selected worksheet or all the worksheets.

Rules while retrieving data from MS-Access

- No need to create a DSN from control panel.
- Input file should have .mdb extension.
- Data types “Number”, “Text” and “Date/Time” from the MS-Access is converted to Numeric, Varchar and Datetime respectively in MySQL.

Rules while retrieving data from Oracle

- No need to create a DSN from control panel.
- Total number of tables created with its records based on schema in which user logs in.
- Datatypes “Number”, “Varchar2” and “Date” from Oracle is converted to Numeric, Varchar and DateTime in MySQL.
• Selected individual table or all tables of a database/schema can be converted.

**Rules while retrieving data from Extensible Markup Language (XML)**

• No need to create a DSN from control panel.
• Input file should have .xml extension.
• Datatype of xml is converted to varchar data type in mysql.

![Figure 4.27: Screenshot of Data Convertor](image1)

![Figure 4.28: Screenshot of Data Convertor](image2)
Figure 4.29: Conversion of .xls extension file into MySQL database
4.3 ALGORITHM FOR N-ELIDB

Input: User Inputted Query/sentence in English language (figure 4.7).
Output: Execution of SQL query with appropriate output or message on GUI

Step 1: Search user inputted natural language query from knowledge base

(figure 4.8)

If search found then

Go to Step 11,

Else

Continue from step 2.

End If

Step 2: Do while for all user tokens

Step 2.1 Perform morphological analyses (table 4.1).

Step 2.2: Remove noisy words or stop words from a sentence

(table 4.2).

Step 2.3: Convert tokens into base token (table 4.4, 4.7, 4.8 and 4.9).

Step 2.4: Call Spell check algorithm (figure 4.3, 4.4).

End do

Step 3: Compare the keywords for DDL, DML processing such as Create, Insert, Update, Delete and Drop (For DDL and DML syntax, refer table 4.10).

Step 3.1: If keyword ‘not’ found then

Continue from step 4.

End If

Step 3.2: If ‘Create’ keyword found then

Call CreateTable method (figure 4.23).

Go to Step 3.8.

End if

Step 3.3: If ‘Insert’ keyword found then

F= call Insert method (figure 4.22).

Go to Step 3.8.

End If
Chapter 4: Analysis & Design of Natural Language Query Interface

Step 3.4: If ‘Update’ keyword found then
   
   F = call Update method (figure 4.22).
   
   Go to Step 3.8.
   
   End If

Step 3.5: If ‘Delete’ or ‘Drop’ keyword found then
   
   F = call delete1 method (figure 4.22, 4.23).
   
   Go to Step 3.8.
   
   End If

Step 3.6: If ‘Select’ keyword found then
   
   F1 = call retrieve method (figure 4.24).
   
   End If

Step 3.7: Call Any2MySQL method (section 4.3.2.1) which converts data
from notepad, excel, sqlserver, oracle, xml to MySQL relational
database.

Step 3.8: Connect to MySQL relational database.

   Step 3.8.1 Select ‘Test’ database.
   
   Step 3.8.2 If (F) then
       
       Execute the DDL and DML statements.
       
       End If

   If (F1) then
       
       Get metadata from table and execute the select
       statement.
       
       End If

Step 3.9: Display SQL output on GUI for successful execution of DDL and
DML Statements or Display appropriate message on GUI
(say, record not inserted, etc).

   Step 3.9.1: Exit

Step 4: Compare for domain (figure 4.6)

   If ‘student’ domain found then
       
       Continue from step 5.

   Else
Display appropriate message on GUI (say, domain not found).

End If

Step 5: Perform Lexical analysis (section 4.2.1.3).

Step 6: Perform Syntactic analysis (section 4.2.1.4).

Step 7: Perform Semantic analysis (section 4.2.1.5).

Step 8: Generate SQL statement for SELECT clause (for SELECT syntax-refer table 4.10).

Step 9: Execute SQL statement for SELECT clause using MySQL relational database.

Step 10: Insert the user query and SQL statement into knowledge base (figure 4.8).

Step 11: If SQL statement successfully executed then

    If record exists then
        Display SQL statement output on GUI.
    Else
        Display the appropriate message on GUI
        (say, record not found).
    End If

End If

End If

Step 12: Exit
4.4 SUMMARY

The chapter presents the proposed architecture for the development of N-ELIDB system. It discusses in detail various algorithms and methodology used in the Linguistic component and Database component. Particularly, it discusses Morphological Analysis (query pre-processing and context resolution, n-gram word, stop words, spelling check, domain mapping, knowledge base), Lexicon Analysis (rules for identifying token and metadata for identifying attribute token), Syntactic analysis (Stanford parser, multi-liaison algorithm), and Semantic analysis (WordNet, lexicon semantic representation, proper noun resolution) of Linguistic component. Finally it discusses SQL Query Generator (intermediate query representation, DDL, DML and SELECT statement), SQL Query Execution (database adaptor, Any2MySQL tool) of Database component.