8-2012

A Semantic Approach for Keyword Search on Relational Databases

Rajvardhan Patil
University of Nebraska at Omaha

A Semantic Approach for Keyword Search on Relational Databases

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Major: Computer Science

University of Nebraska at Omaha

by

Rajvardhan Patil

August, 2012

<u>Supervisory Committee:</u>

Dr. Zhengxin Chen

Dr. Parvathi Chundi

Dr. Peter Wolcott

A Semantic Approach for Keyword Search on Relational Databases

Rajvardhan Bhaskar Patil, M.S.

University of Nebraska, 2012

Advisor: Dr. Zhengxin Chen

Abstract

Today's search engines make it easier for the user to browse and query the online available data. But when it comes to structured data, the queries have to be structured too, in order to retrieve the data. This makes it difficult for novice users, with no knowledge of the underlying schema or query language, to access the relational data. Therefore, to query the structured data in an unstructured language of web, there is a need to map the user keyword queries to their equivalent SQL format. This research is intended to bridge the gap by introducing a framework named STRUCT. Unlike most of the existing work which pays very little attention to the contextual information provided by the user, our approach takes these details into account to elucidate the implied structural information necessary for constructing the SQL clauses. One fundamental issue on keyword search in traditional databases is how to interpret users' information needs behind keywords they provided. A common approach of many prototype systems is to make such interpretation as a designer's choice (such as imposing AND or OR semantics, or a combination), leaving no choice to users. A much more meaningful approach would be allowing users themselves to specify the required semantics through contextual information. So can we build a system which stays with the simplicity of Keyword search, yet can incorporate the contextual information provided in the user query? STRUCT answers this question by

taking English language queries involving intended keywords. Instead of resorting on a full-fledged natural language processing, the unneeded words in the queries are discarded. Only the specific contextual information along with the keywords containing database contents will be used to construct SQL queries. The contextual information is used to interpret the meaning of the queries, including the semantics involving AND,OR and NOT. In this thesis we describe the architecture of STRUCT, procedure of English query processing (parsing), basic idea of the grouping algorithm, SQL query construction and sample results of experiments.

Acknowledgement

I am extremely grateful and indebted to my thesis advisor, Dr. Zhengxin Chen, for accepting me as his student. His constant guidance, encouragement, and support throughout the completion of my work were the key for my thesis. I owe a special thanks to Dr. Peter Wolcott for his invaluable guidance and enlightening discussions on the topic which made this thesis possible. I am also grateful to Dr. Parvathi Chundi for providing valuable feedback and comments on my thesis.

I would like to thank all the professors of the Computer Science department at University of Nebraska - Omaha, who taught great courses. I would also like to thank several students in our research group, and my friends for their support, motivation, and help during my stay here at Omaha.

Most importantly, I would like to thank my mother and father, Asha and Bhaskar Patil, and brother Ranjeet, who have cared for me continuously, in every possible way, from halfway around the world.

Table of Contents

1. **Introduction** 1

1.1. Contributions 6

1.2. Outline of Thesis 6

2. **Overview** 7

2.1. High Level Architecture 8

3. **Query Parsing** 10

3.1. Inverted Index 10

3.2. Thesaurus 11

3.3. Delimiters 12

3.4. Divide and Conquer Strategy 15

3.4.1. Divide Phase 16

3.4.1.1. Grammar 16

3.4.2. Conquer Phase 17

3.4.3. Example 18

3.5. DFS and Schema Graph 20

4. **SQL Query Construction** 22

4.1. Association Data-Structure (ADS) 23

4.2. Resolving the Ambiguity 24

4.3. Handling of "NOT" Operator 25

4.4. Grouping Algorithm 26

4.4.1. Attribute Domain Information (ADI) 27

4.5. Link Data-Structure (LDS) 36

4.6. Possible Combinations 37

4.7. Constructing SQL Clauses 39

5. **Experimentation And Evaluation**..41

5.1. Sample Results 41

5.2. Performance Evaluation 43

6. **Related Work** 47

7. **Conclusion and Future Work** 50

REFERENCES..52

List of Figures

Figure1.1: Banking Schema.....4

Figure2.1: Stages in STRUCT.....8

Figure2.2: High Level Architecture.....9

Figure3.1: DFS Graph Example.....21

Figure4.1: Grouping Algorithm.....30

Figure5.1: Screen Shot of Sample Query-23.....41

Figure5.2: Screen Shot of Sample Query-24.....42

Figure5.3: Screen Shot of Sample Query-25.....42

Figure5.4: Impact of Specified Attributes.....45

List of Tables

Table3.1: Attribute_info....11

Table3.2: Table_info....11

Table3.3: Employee....11

Table3.4: Thesaurus....12

Table4.1: ADS....23

Table4.2: LDS....37

Table4.3: Unorganized format for Query 22....38

Table4.4: Unorganized format of Cell....39

Chapter 1

Introduction

More and more organizational websites are relying on structured database to answer the user queries. And hence to satisfy the user needs and to make the search easier, websites are providing HTML forms, replicating the underlying database schema, where user is asked to enter the values for the columns in the database tables. This task can sometimes be laborious since user may have to go through many such forms to reach the desired result. This also limits user's freedom by restricting him to the static and predefined queries handled by the forms. Ample amount of research has been done in this field, to break the bonds and limits set down by these static search systems.

On the other hand, Information retrieval (IR) on the web documents not only gives the end-user complete freedom of search but also keeps him/her unaware of the underlying data representation and association. But this technique of IR on web documents cannot be applied on relation data. Since, in relational database, the data that user is looking for, may not be necessarily located in a single table. Due to normalization, the required user data might have been scattered across several tables. So in order to come up with meaningful result for the keyword search over relational database, we need to know the data that user is looking for, where that data is located in the database and then find out a way to connect it through the foreign key links. Most of the research techniques are able to execute these operations, but apart from it, these techniques have their own limitations as discussed below.

Research on keyword search in database community has achieved a lot of success, as exemplified in a number of prototype systems, such as DBexplorer, Discover, BANKS, etc. [1,2,3,9,10,11,12 13, 14,15,16,17,18,20]. Recent developments have been nicely summarized in several tutorials, surveys and comparative studies [4,5,6,8,21]. Research areas of interests have been moved from keyword search in relational databases to various advanced issues such as keyword search in multimedia data and data streams, as shown in the topic of interests in Keys 2012 Workshop (http://datasearch.ruc.edu.cn/keys2012/index.html). Yet, many fundamental issues on keyword search in traditional databases remain. One such issue is how to interpret users' information needs behind keywords they provided. A common approach of many prototype systems is to make such interpretation as a designer's choice (such as imposing AND or OR semantics, or a combination), leaving no choice to the users. A much more meaningful approach would be allowing users themselves to specify the required semantics; however, supporting such full-fledged natural language queries would incur a significant amount of processing burden on the system. Therefore, it is worth exploring a "middle-ground" approach, namely, instead of limiting user inputs to keywords alone, how about building a system which allows users enter English language queries containing the desired keywords, so that the contextual information found in English queries can be used to interpret query semantics and form structured database queries? We believe this is possible because we can take advantage of the structural information implied in database schemas.

In this thesis we introduce our experimental system STRUCT to explore this idea. STRUCT takes English language queries involving intended keywords; by doing so, it supports a kind of "expanded keyword search". In this thesis, we further refer to such search as English query search. Instead of resorting on a full-fledged natural language processing, the unneeded words in the queries are discarded. Only the specific contextual information along with the keywords containing database contents will be used to construct SQL queries. Contextual information is used to interpret meaning of queries, including the semantics involving AND, OR and NOT. Note that such an "expanded keyword search" only makes sense for *structured* data, because without any meta-data information of underlying database, it is impossible to distinguish keywords from contextual information (namely, all are treated equally as keywords, as in Google search). When the underlying data is structured then all the implicit information provided through the rich meta-data, structural format of data, normalization of the tables, mapping between the tables, and association of the values with the attributes can be collectively used to make the search engine understand the semantics of the structured database. Furthermore, in order to allow users express semantics of the queries, an interface allowing users entering simple natural language queries containing intended keywords should be supported. The contextual information found in the English queries can be then used to derive the user query semantics and hence mapped it with the database semantics in order to retrieve the desired result. So far only few authors have addressed the problem of keyword query interpretation, such as [8], where query interpretation is conducted at a fairly late stage, only concerned with ranking. In fact, interpretation of keywords provided by the users should be conducted at early stage so that their intention can be incorporated into SQL query construction. But just interpreting the keywords in the query will not suffice, since to fully explain the information need, the user has to enter natural language query containing the contextual information along with the intended keywords. Another issue as indicated in the recent ICDE 2011 tutorial [21] is that there is a significant challenge in resolving structural ambiguity, since keyword queries are structure-less. A much more meaningful approach to resolve the ambiguity would be allowing the users' to specify contextual information, which would decipher the hidden or implicit structure of the query.

In this thesis we describe STRUCT, which processes the given user query not just to locate and connect the tuples having query data, but also to identify the contextual information of the query needed to deduce the implied meaning. It further interprets and incorporates the derived semantics in the SQL statement to be constructed. STRUCT levitates the physical model of database by representing it in a undirected graph, where the relational schemas are represented as nodes and the foreign key links are represented as edges. Depth first search technique is used by STRUCT to traverse the schema graph and further construct the sub-graph (candidate network), representing the path between relations having the query keywords. As an example, schema graph of the well-known banking database from [19] is shown below in Fig. 1.

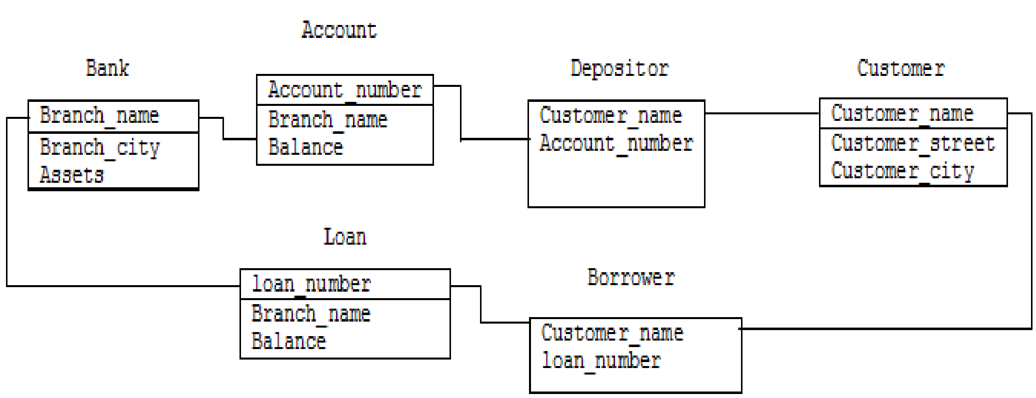

Figure 1.1 Banking Schema

Consider a query that STRUCT can execute for the above schema:

Query 1: "Find customers who have an account in a branch called Brighton".

With the metadata information obtained at the time when the database was loaded, and with the help of the built-in thesauri, STRUCT maps the keywords customer, account and branch keywords in the query to their respective attribute names in the schema, namely: customer_name, account_number and branch_name. These attributes, that user is looking for, forms the SELECT clause of the SQL statement. STRUCT further looks for the involved tables in join computation so as to cover all the attributes in the query and hence forming the FROM clause. In this example, the minimum number of tables involved is 2: Account, and Depositor. Another combination could have been: Bank, Account, Depositor and Customer; but would have led to more join computations and hence increasing the query cost. In addition, if any attribute-value pair is mentioned by the user, then it is represented as a condition in the WHERE clause. For example, here (branch_name ='Brighton') forms the condition. Additionally, the Primary-foreign key relationship is also added to the where clause. So the final SQL query is:

Select customer_name, account_number, branch_name

From Account, Depositor

Where (Depositor.Account_number = Account.Account_number) and

(brach_name='Broghton');

In later sections we will describe in more detail, how the construction of WHERE clause gets tricky, as the number of involved sub-queries, conditions, operators and operands increase in number.

1.1 Contributions:

The contributions of this paper are as follows:

i. We point out the intrinsic limitation of keyword search in databases due to its lack of dealing with semantics, and offer a simple way of alleviation by exploiting structural information implied in database schemas. Instead of restricting user inputs to keywords only, users can now enter English queries containing these keywords, where the contextual information is used to interpret query semantics.

ii. We present a parser based on Context Free Grammar (CFG), which is able to segregate and hence disclose the implicit structure of the given English query.

iii. We introduce a novel grouping algorithm which is able to group the operands by determining the precedence of involved operators (AND, OR, NOT), and hence revealing the query semantics, needed to construct an equivalent SQL statement.

1.2 Outline of Thesis:

The balance of this paper is organized as follows. In Section 2 we present the general overview of STURCT, which is followed by a description of query parsing in Section 3 and methodology for SQL query construction in Section 4. In Section 5 we present sample experimental results along with a discussion of performance evaluation. In Section 6 we compare STRUCT with related work. Section 7 concludes the paper and provides a discussion on future extension and improvements of STRUCT.

Chapter 2

Overview

In STRUCT, a user is provided with an interface to interact with the underlying relational database, where he can enter the English statement queries, getting back the results in tabular format. This overall process from entering the query to retrieving the results can be briefly categorized into the following four stages: Parse, Fragment, Associate, and Organize.

Initially, the given user query is parsed by STRUCT in order to parenthesize the operands and conditions. The parenthesized query helps STRUCT to know the precedence of the involved operators. In the next stage, the parsed query is broken down into fragments, where every token is further represented with the help of rows and columns. This intermediate tabular representation of the query makes the STRUCT know about the involved values, attributes, tables, and operators in the given query. Now, in order to associate operands and attributes through the comparison operators, and to connect the conditions through the logical operators an unorganized format is constructed (to be discussed later).

Finally, after all the association and linking, an SQL statement comprising of SELECT, FROM and WHERE clause is built from the unorganized format. The derived SQL query is then executed on the structured database, to display the results back to the user. These stages are depicted in Figure 2.

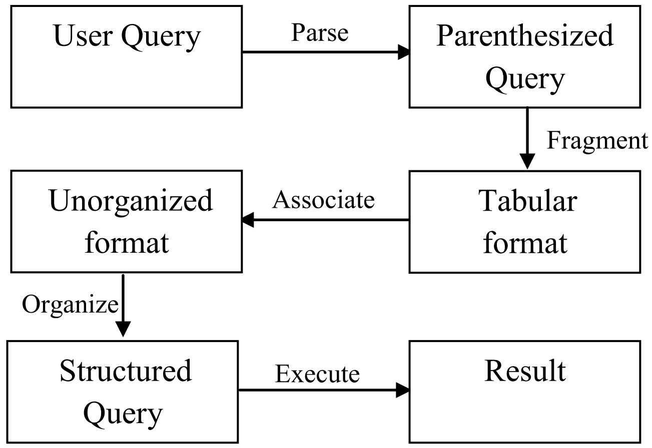

Figure 2.1 Stages in STRUCT

2.1 High Level Architecture

STRUCT provides several built-in components some of which are independent to any database, like thesauri and delimiters list. Before performing keyword searches on a particular database, STRUCT loads that database into the system and does the offline crawling to build an inverted index for the lookup purpose. It also makes the use of built-in list of delimiters and thesauri to parse the given user query. Given a user query, STRUCT comes up with an equivalent structured query, executing it on the underlying database, to retrieve the results in an efficient manner. Building such query search requires the following: (a) A preprocessing step called *offline crawling* to build an inverted index; (b) Parsing step used to divide the given user query and then classify the query keywords as data and meta-data; (c) Grouping the values based on the logical operators in the query; (d) Structuring the user query into its equivalent SQL format; (e) optimizing and executing the derived SQL query to fetch the result back to the user. A high level representation of the architecture STRUCT, used to construct an SQL statement is shown in Figure 3. More details of each component are given in consecutive sections.

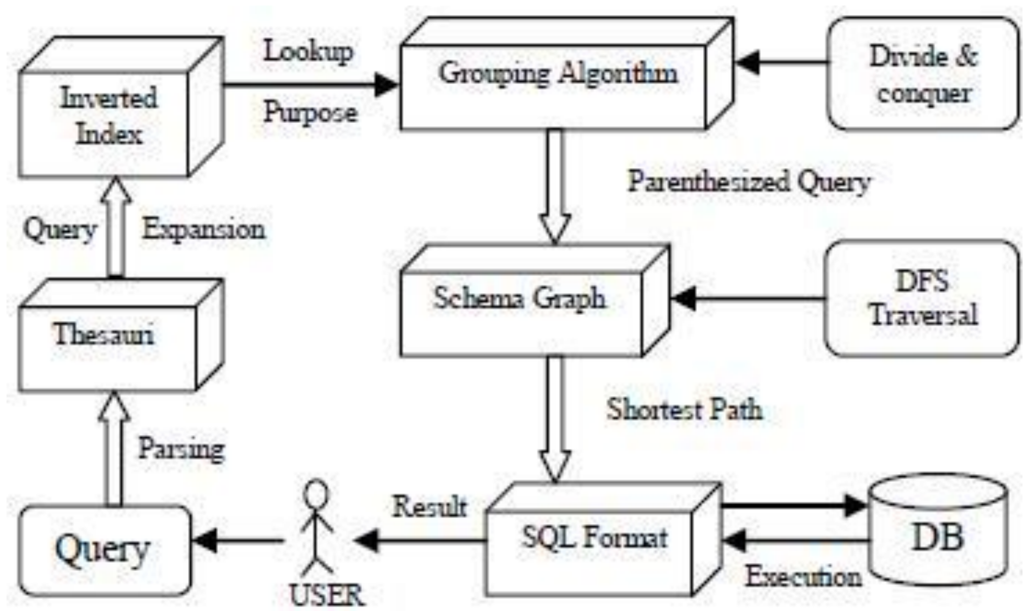

Figure 2.2 High level Architecture

Chapter 3

Query Parsing

Before going into the details, we look at some of the important concepts, components and algorithms used by the STRUCT. English statement queries, submitted by the user, always have implicit structure associated with them. STRUCT makes use the concepts of *delimiters* and *attribute domain information (ADI*) to reveal the hidden structure and to interpret the intended purport of user queries. Inverted index and thesauri components guide STRUCT for keyword lookup, and to further expand and parse the user query. The grouping algorithm, based on divide and conquer strategy, assists STRUCT in grouping the operands and in deciding the precedence of the operators. STRUCT employs depth first search algorithm to traverse schema graph of the database, in order to determine the association of the user data. Details of components in query parsing are discussed below.

3.1 Inverted Index

A database is enabled for query search by building an inverted index on it. The inverted index rephrases the relational database by associating every value to its corresponding column name and table name. This helps STRUCT to locate the meta-data information for given query keyword. Our system trades storage space and offline indexing time to significantly reduce the overall query computation time.

Inverted Index consists of two tables: Attribute_info and Table_info, with examples shown below.

Table 3.1 Attribute_info

Value	Attribute_list
Green	Tea, lantern, Color,streelight ...
Pittsfield	Branch, city, capital…
…	…

Table 3.2 Table_info

Attribute	Table_list
Customer	Bank, client,buyer,...
Balance	Account, Loan, …
…	…

The Attribute_info table gives the attribute information of where the value is located. This information can be further used to identify the tables associated with those attributes by using the Table_info table. Thus by using these two tables, representing inverted index, one can easily associate the value with the corresponding attributes and table names.

3.2 Thesaurus

While constructing an English statement query, user cannot be restricted to the terminologies comprising of attributes and table names. For example, if the database has one of the following tables:

Table 3.3 Employee

Employees	Address	...
James	Omaha	
Tom	Lincoln	...

Now, consider the following queries:

Query 2: Find employees having Omaha in the address field

Query 3: Find employees living in Omaha

Note that Query 2 is easily parsed by STRUCT by using inverted index, as it can note down the attribute “address” and associate it with the value “Omaha” involved in the query. But in query-3, address’ synonym “living” is used by the user to represent the meta-data information. So to parse such queries, STRUCT refers in-built thesaurus and replaces the user keywords with the meta-data information, wherever necessary. A sample thesaurus table along with its schema is shown in Table 4.

Table 3.4: Thesaurus

Meta-data	Related-words
Employees	Worker, laborer, staff,
Address	Living, Location,
…	…

3.3 Delimiters

User query usually consist of the information that he/she is looking for; but sometimes the user may provide additional information through conditions, in order to specify their requirements and to narrow down the result. Now to deal with the given user query, STRUCT breaks the query into sub-queries with the help of delimiters. Below we describe the delimiters of a query.

A user query can comprise of one or more sub-queries, where each sub-query represents a group of dependent conditions. These conditions within a sub-query are connected with logical operators. Whereas, each time when an independent condition need to be specified, a new sub-query is initiated within a query. Now in order to locate the beginning of each such sub-query, we need the help of delimiters.

Delimiters: *Delimiters* are the words used by the user to connect different sub-queries formulating into a query.

STRUCT identifies the delimiters on the basis of rules used forthe construction of the English sentence. An English user query comes with a subject and a predicate. The subject is the information that the user is looking for; and the predicate tells us something about the subject‘s requirement with the help of sub-queries (SQ). These requirements usually consist of subject‘s attributes (objects), their values, and operators; collectively representing the condition. In SQL, the subject of the English query represents the SELECT clause, and the predicate constitutes the WHERE clause. FROM clause helps us to locate and connect query keywords, in the underlying database. Now, in order to link one or more sub-queries with the subject, in a possessive manner, user needs a word that can represent the subject as well as say something about the subject. In English natural language, this simultaneous role of being a noun and a verb is done by Gerund.

Here is an example where a gerund is the delimiter:

Query 4: A Toyota car **having** Red color and production year > 2000 or **giving** mileage of 30 miles per gallon.

Subject: A Toyota car

SQ 1: Red color and production year > 2000 or

SQ 2: mileage of 30 miles per gallon

The above query and most of the query examples in sections to come, are against the car database, having the following relational schema:

Manufacturer(MID, Make, category)

Models(Model_name, style, MID)

Vehicle_details(Model_name, style, manufacture_year, color, price, doors, gears, mileage)

In certain circumstances the gerund in the query might represent a value or an attribute. In such cases, inverted index helps STRUCT to classify it as a value or meta-data, and not as a delimiter. For example, consider again Query 3:

Find employees living in Omaha

Now the thesaurus (as shown in Table 4) helps to figure out that the "living" word represents the attribute "address", and hence will not be treated as a delimiter.

In Addition to gerunds, even the pronouns and helping verbs helps the predicate to get associated with the subject in question, and hence playing the role of delimiters.

Examples, where the helping verb is the delimiter:

Query 5: Check for the Students getting GPA < 3.0 and **were** absent for more than 10 days.

Subject: Check for the Students

SQ 1: GPA < 3.0 and

SQ 2: absent for more than 10 days

Examples, where interrogative and relative pronoun are the delimiter:

Query 6: Find a car **which** is red in color and price < \$3000 or **whose** mileage > 20

Subject: Find a car

SQ 1: red in color and price < \$3000

SQ 2: mileage > 20

Most frequently used pronoun delimiters are {who, whose, which, whom, what, that, ...} and so on. Some prepositions (not limited to) like "by, with" also act as delimiters. Let's see the examples, where preposition is the delimiter:

Query 7: Look for a book **by** author xyz or abc **with** pages no less than 100

Subject: Look for a book

SQ 1: author xyz or abc

SQ 2: pages no less than 100

In certain circumstances, prepositions like "in", or helping verb like "is", does not act as delimiters. For example, Red *in* Color; mileage *is* greater than 40. In these examples, rather than acting as delimiters, they associate the attribute with the value. To make a distinction between this duality, STRUCT associates a flag while parsing, and checks whether the words appear in between the attribute and value; if not, they are treated as delimiters. Gerunds, prepositions, helping verbs, and pronouns because of their roles, help STRUCT to decide the beginning of a new sub-query. Based on these delimiters STRUCT breaks the query into sub-queries and then into conditions.

3.4 Divide and Conquer Strategy

This strategy comprises of two distinct phases: divide phase and conquer phase. The divide phase makes the use of context free grammar (CFG) to divide the given user query, whereas, the conquer phase makes the use of grouping algorithm to parenthesize the given query. But before proceeding into these phases, the given query is "skimmed" in order to retain just the needed information such as attributes, values, conjunctions, delimiters, comparison and logical operators; any other information contained in the original English query is discarded by the search engine.

3.4.1 Divide Phase

As noted, English query comprises of subject and predicate, where predicate constitutes of one or many sub-queries. To add further, each sub-query is then divided into conditions and finally into attributes, and values. Since STRUCT only pays attention to contextual information helpful in query construction, rest of the irrelevant words in the user query will be discarded. Therefore, no rigid requirements of user queries are required. The overall Grammar used by STRUCT, to divide the given user query, can be summed up as follows:

3.4.1.1 GRAMMAR

STRUCT makes use of a CFG grammar to interpret the user- submitted English queries. In essence, each English query is made of sub-queries, which are conditions connected through logical operators. More formally, STRUCT makes use of grammar G = (N, Σ, P, Q), where the non-terminal symbols are: N= {Query, Subject, Predicate, Sub-query, Condition}; the terminal symbols are: Σ= {attribute, value, first-delimiter, subsequent-delimiter, logical-operator, comparison-operator, wrapper}; Query is the starting symbol and the set of production rules (P) are:

i. Query → Subject (first-delimiter Predicate)*

ii. Predicate → Sub-query (subsequent-delimiter Sub-query)*

iii. Subject → Condition (logical-operator | Condition)*

iv. Sub-query → Condition (logical-operator | Condition)*

v. Condition → (attribute comparison-operator value | value comparison-operator attribute)*

vi. Condition → (attribute)* (value wrapper)* (value | attribute)*

("wrappers" are defined later in section 4).

Since the subject represents the information that user is looking for, we assume that it appears at the start of the English query (which is a fair assumption to be made), and hence make the use of first-delimiter, to separate it from the predicate. In the above grammar, subsequent delimiters are used to connect sub-queries, and logical operators to connect conditions. Further in a given condition, comparison operators are used to associate attribute with value; forming an attribute-value pair. If not explicitly specified, the comparison operator is set to '=' by default.

3.4.2 Conquer Phase

In conquer phase, Attribute domain Information (to be discussed later), helps to parenthesize the given user query, letting us know its implicit structure. In this phase, the grouping algorithm is recursively used to parenthesize the operands, followed by conditions and then the sub-queries and eventually the query. Note, when all the sub-queries are grouped to form a predicate, then eventually the subject is associated with predicate to build a query. We will discuss more about this phase in subsection 4.4.

3.4.3 Example

Below, we illustrate an example, to explain the execution of Divide phase based on CFG. In the following example, the grouping (conquer) phase needs further explanation, which is elaborated in subsection 4.4; where the grouping algorithm and ADI are detailed. Consider the following example: Query 8: find a Honda or Toyota car with 2 doors and Color Red or which gives mileage greater than 20 miles per gallon.

Discarding non-essential information:

STRUCT examines the English query to retain only the required information.

Resulting Query: Honda or Toyota car with 2 doors and Color Red or which mileage > 20

Divide:

Now the delimiters in the parsed query are: {with, which} So based on rule (i) of grammar, we divide the query into subject and predicate, with respect to the first-delimiter: "with"

Subject: Honda or Toyota car Predicate: 2 doors and Color Red or which mileage > 20.

Now based on rule (ii) of grammar, the predicate is divided into multiple sub-queries, depending upon number of delimiters it has. In the above example, we have just one subsequent-delimiter: "which'. So the resulting sub-queries (SQ) are:

SQ-1: 2 doors and Color Red or

SQ-2: mileage > 20

In the subject, the 'or' acts as a wrapper. Later in section 4.2 we will explain why it acts a wrapper and not a logical-operator. Therefore, by using grammar rule (iii) and (vi), we can formulate only one condition out of the subject.

Subject: Condition 1: Honda or Toyota car

Now based on rule (iv) of grammar, the sub-queries are further divided into conditions. In sub-query 1, the "and" acts as a logical-operator to separate the conditions, as following:

Sub-query 1: Condition 2: 2 doors , logical-operator: 'and'

Condition 3: Color Red, logical-operator: "or"

Sub-query 2, has only one condition with no logical-operator in it, and hence no division takes place.

Sub-query 2: Condition 4: mileage > 20.

These conditions are further divided into values, attributes, and comparison operators, if any. The logical-operators between the conditions are also retained; if not specified, STRUCT defaults it to "and".

Subject: Condition 1: value (v1) – Honda, wrapper – "or" value (v2) – Toyota, attribute – car default logical-operator: "and"

Sub-query 1: Condition 2: value – 2, attribute – doors logical-operator: "and"

Condition 3: attribute – Color, value – Red logical-operator: "or"

Sub-query 2: Condition 4: attribute – mileage, value – 20 comparison operator – " >".

All the above fragmented information is given as an input to the conquer phase, where the grouping algorithm and attribute domain information act together, to rebuilt the query but in parenthesized format. This process is detailed in section 4.4. For the time being, assume that the grouping takes place in the following order.

Conquer: Now, STRUCT parses each condition to determine the comparison operators, and attributes associated with each of the operands. Default comparison operator is set to "=". Subject: Condition 1: (car = Honda) or Condition 2: (car = Toyota) and

Sub-query 1: Condition 1: (doors =2) and Condition 2: (Color = Red) or

Sub-query 2: Condition 1: (mileage > 20)

These conditions are further grouped by the grouping algorithm (to be discussed later), to form a parenthesized sub-query. Subject: ((car = Honda) or (car = Toyota)) and Sub-query 1: ((doors = 2) and (color = Red)) or Sub-query 2: (mileage > 20) Now, the predicate consisting of two sub-queries is formed, before associating the subject with it. Predicate: (((doors = 2) and (color=Red)) or (mileage > 20)). Eventually the subject is associated with the predicate to form the entire parenthesized query. In the example, as no logical operator was used to associate the subject with predicate, STRUCT uses "and" as the default one, to connect subject and predicate. Parenthesized format: [((car = Honda) or (car = Toyota)) and (((doors = 2) and (color=Red)) or (mileage > 20))]

Grouping algorithm makes the use of the above divide and conquer strategy to parenthesize the query.

3.5 DFS and Schema Graph

As discussed, STRUCT represents the given database as a undirected schema graph, where the nodes are the relational schema and edges are the foreign links. Now, this graph is used by the STRUCT to check whether the query keywords are associated or not. This association is derived from the primary-foreign key relationship. If the attributes representing the keywords are connected through edges, then it implies that the keywords are associated with each other.

Before the traversal begins, the tables (nodes) having at-least one user query keyword (representing value or meta-data) are treated as destination nodes. From these destination nodes, iteratively one of the nodes is selected as a source node, to form a path covering the rest of the destination nodes. Such paths, connecting all destination nodes, are further used for building the SQL statement. While constructing a path, care is taken to avoid formation of cycle, as it may result in redundant join computations; whenever a node is traversed, it is marked as visited, to avoid further visits.

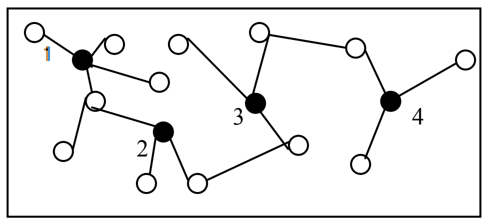

Figure 3.1 DFS Graph Example

In the above diagram, the numbered (black) nodes indicate the destination nodes, where the keyword information is encountered. The graph is traversed by using Breadth First Search technique to link the required nodes resulting into a path. STRUCT constructs all the possible paths, connected by primary–foreign key edges, making sure that all the user specified values are covered. These paths help us to know the tables involved in the join computation. Eventually to optimize the execution, the path having least number of involved relations (nodes) is selected.

Chapter 4

SQL Query Construction

A user query consists of values (data), attributes (meta-data), conjunctions, operators (logical or comparison), wrappers and delimiters. *Wrappers* are the logical operators used by the user to specify one or multiple values for a given attribute. Comparison operators are used by the user to form attribute-value pair or to specify the range limit for the attributes, whereas, logical-operators are used by the user to connect multiple conditions. In addition, a user may use delimiters to join multiple sub-queries, formulating into a query. STRUCT goes through the following steps, in order to represent the above user specified information structurally:

i. Parsing the given query

ii. Parenthesizing the refined query

iii. Forming Tabular representation for the parenthesized query

iv. Constructing Unorganized format from tabular representation

v. Building an equivalent SQL statement of given English query

Initially, STRUCT parses the given query to classify the information into various categories (as listed down by STRUCT‘s CFG grammar). By applying the grouping algorithm, STRUCT parenthesizes the entire query to deduce the implied semantics of the given user query. In order to derive an SQL query out of it, the parenthesized query is further represented in tabular format and then into unorganized format to eventually come up with an equivalent SQL statement.

Overall, while transiting through the above steps, the intermediate results are represented by constructing the following two data structures. Firstly, to associate attributes with values through comparison operators, Association data-structure (ADS) is constructed. Secondly, to represent the conditions connected through logical operators, Link data-structure (LDS) is created. The delimiter information is implicitly carry forwarded through the parenthesis used to group the query. More details are given below.

4.1 Association Data-Structure (ADS)

SQL statement generation requires the list of tables and columns where the keywords may occur to be known. Inverted index helps STRUCT to satisfy this basic requirement. Using the Inverted index, STRUCT is able to classify the keyword as either a value or attribute. If not found, it consults the thesauri to check if the user has used any synonym for the meta-data (attribute) information. If the user has specified a value along with the corresponding attribute information then such attribute-value pair is associated with the help of ADS. Comparison operators, if encountered, are also noted down. Based on this information, STRUCT builds the ADS data structure from the given user query, as illustrated in Query 11.

Query 9: Find a car which is Red and having price< 7000

Table 4.1 ADS

Attribute	Comparison	Values
Car	=	-
-	=	Red
Price	<	7000

[Note: by default, comparison operator is set to “=”, unless stated]

This ADS links the values with their corresponding attributes by using the comparison operator. If any of the information is missing then its kept blank (-). Only the information specified by the user is utilized to construct this data structure. ADS information is later used by STRUCT in constructing LDS.

Once the ADS is built, the next steps are to:

i. Resolve the ambiguity raised by “AND, OR”

ii. ii. Handling of “NOT” operator

iii. iii. Determine the Precedence of the logical operators

We will look at each of these issues in the following subsections.

4.2 Resolving the Ambiguity

Here, STRUCT checks whether the “AND, OR” act as wrappers or logical operators, in order to resolve any possible ambiguity. As stated earlier, wrappers are generally used by the user to specify multiple values for the same attribute (domain). For example, find a car which is red OR blue OR green in color. Here the color attribute is used to capsulate the three values (red, blue and green). In such cases, the “AND, OR” should be treated as wrappers and be grouped together. In addition, when “AND” is used as a wrapper, it often represents “OR” semantics. Since it is not possible to specify multiple values for the same attribute unless it‘s connected by “OR”. For example, find customers living in Omaha AND Lincoln. Here user is looking for customers who either live in Omaha OR in Lincoln but not at both the places. Another example could be, find cars having red AND green color; which implies, color = red OR color = green. Only if the attribute is multi-valued, should the semantics of “AND” be preserved.

Query 10: Find a car having color White **or** Black **and** price < $3000

Here, the value "Black" acts as an operand to both "OR" and "AND". But, the attribute domain of White and Black are the same, therefore "OR" acts as a wrapper. As per the grouping algorithm, the wrappers are given higher preference than logical-operators; so Black is grouped with White and not with "$3000". Therefore after grouping, we have: (White or Black) and ($3000) If "AND" acts as a wrapper, then as stated, replace it with "OR" in order to group the operands. An example is given in Query 13. Query 11: Find a car having red AND green color Parenthesized format: find a car having (red or green) color Here, the operands "red" and "green" belong to the same attribute domain; so the "and" is replaced by "or" and then the operands are grouped together. On the other hand, if the "AND, OR" act as logical-operators, i.e., if involved values have or belong to different attributes, then the grouping algorithm is used to determine the precedence of those logical operators.

4.3 Handling of "NOT" Operator

As compared to logical "AND, OR" operator, in STRUCT, the "NOT" operator is handled in a distinctive way. The "AND, OR" play the role of either the wrapper or the logical operator, whereas, "NOT" either acts as a delimiter, or as a unary operator. The following example illustrates the difference:

CASE A: ~ (A operator B) Here, NOT acts as a delimiter to form a condition Query 12: Find a car which does not have mileage < 20 and price > 20000 parenthesized format: ~ (mileage < 20 and price > 2000)

CASE B: (~A operator ~B) Here, NOT acts as a part of operand (unary operator).

Query 13: Find any car but not Honda and should not be Red in color parenthesized format: ~ (Honda) and ~(Red)

As we will see in the algorithm, grouping in CASE A is not based on the operator, whereas, grouping in CASE B depends upon the operator at hand.

4.4 Grouping Algorithm

Frequently a user may enter a query which has more than one logical operator. In such case, it becomes essential to determine the precedence of the involved logical operators. Since if not grouped properly, the same query can lead to multiple interpretations. So it is not only important to determine the precedence of the operators involved, but also to match it with the user intended semantics. Consider the following example: V1 and V2 or V3 [V1, V2, V3 represent values]

In such a case it becomes difficult to determine whether the operand 2 (V2) belongs to operator 1 (and) or whether it belongs to operator 2 (or). There can be two different interpretations of the above query:

i. (V1 and V2) or V3

ii. V1 and (V2 or V3)

If more operators are involved, then the number of possible combinations would manifold. Now the task is to determine one that matches the user semantics. To do so, it is important to understand how the user, without any use of parentheses, groups the operands with the operator. The ADI associated with the user query, does implicitly specify the precedence of the operators needed to parenthesize the given user query.

4.4.1 Attribute Domain Information (ADI)

Attribute domain information lists the attribute of the respective operands. Prior to grouping any of the operands, their corresponding attribute information is checked. Based on this attribute information and the involved operator, the grouping algorithm decides the grouping of the operands. Furthermore, the attribute domain information is updated through a recursive process, as shown below. In grouping algorithm, the grouping takes place at four levels: a) operand level b) condition level c) sub-query level and d) query level. A condition is formed whenever two operands are grouped.

These conditions, in turn, act as operands for the next grouping cycle. Conditions are further grouped to form a sub-query. For the next iteration of the algorithm, these sub-queries are treated as operands. This cycle continues until all the possible groupings in the query are not parenthesized. To make the above grouping of operands possible, the attribute domain information is kept updating at every level of granularity. At each grouping level, the attribute domain information of the previous level is used to decide whether the operands can be grouped or not. . In addition, whenever the grouping of operands takes place, their corresponding attribute information is also updated to reflect the grouping. Based on this new attribute domain information, the next level grouping is decided, and so on. Consider the following example:

Query 14: Find cars with white color and 2 doors or having 4 doors and Black in color.

For this example, we focus more on the ADI part and its amendments. The working of grouping algorithm and its grouping decisions will be discussed later. For the time being, suppose the grouping algorithm groups the operands in the following order. Note here, how the attribute domain information gets updated.

i. white and 2 or 4 and Black [Operand information] color and doors or doors and color [ADI]

Grouping algorithm uses the above attribute domain information, to group the operands in the following manner:

ii. Group operands White, 2 (white and 2) or 4 and Black [Operand information]

(color and doors) or doors and color [ADI]

Here, the corresponding attribute domain information is also grouped. Now, the grouping algorithm uses the above updated attribute domain information, to group the operands in the following manner:

iii. Group operands 4, Black

(white and 2) or (4 and Black) [Operand information]

(color and doors) or (doors and color) [ADI]

The attribute domain information is again updated to reflect the grouping of operands.

iv. Group the 2 Conditions ((white and 2) or (4 and Black)) [Operand information]

((color and doors) or (doors and color)) [ADI]

In the above example, the attribute domain information of the previous step decides whether the grouping of operands, in the next step, is possible or not. Once decided, it itself gets updated to reflect the changes.

Two important properties associated with ADI are:

Property 1: The attribute domain information connected by "and, or" logical operators is commutative in nature. i.e., changing the order of involved attributes does not change the semantics.

i.e., (A1 and A2) = (A2 and A1)

Example: (doors and color) = (color and doors) similarly, (A1 or A2) = (A2 or A1)

Example: (model or price) = (price or model)

Property 2: The duplicated attribute domain information connected by “and, or” wrapper is removed, such as: (A1 and A1) = A1.

For example: (doors and doors) = (doors).

Also, (A1 or A1) = A1.

For example: (colors or colors) = (colors).

We now map the user‘s un-parenthesized way of structuring the query to the parenthesized format by using the grouping algorithm. The sub-queries formed by the divide phase of CFG grammar along with its ADI information, is given as an input to the Grouping algorithm, shown in Figure 4. The algorithm recursively groups the operands and then the conditions, simultaneously updating their corresponding ADI information, until the entire sub-query is not parenthesized. Further, all the grouped sub-queries along with their ADIs, is given as an input to the grouping algorithm, to check whether further grouping across sub-queries is possible or not. This helps us to formulate the entire predicate in parenthesized format. Eventually, the subject is associated with predicate to come up with a complete parenthesized query. The resulting format helps us to determine the precedence of the involved operators over one another.

Here are a few notational remarks used in Figure 5:

A – left operand,

B – right operand,

and (parentheses) – implies grouping

GROUPING ALGORITHM:

Algorithm: Grouping Algorithm
Input: un-parenthesized sub-query
Output: parenthesized sub-query
Procedure:

a. Capsulation procedure is used to note down the conjunctions, if any, in the query.
b. Conjunctions are given higher priority than Operators.
c. As per the precedence rule, the logical operators are prioritized in the following order: NOT, AND, OR
d. If the same operator is repeated, then the priority is determined through Associativity rule: the left hand side operator is dealt first.

Grouping Algorithm satisfies the above priority rules, by building the following 1 to 10 different cases. In each case, grouping of the operands takes place based on the attribute domain information (ADI) and the operator at hand:

(A) Group ALL the conjunctions in the query.

If ADI [A] = ADI [B]
1. if "AND" then (A or B)
2. if "OR" then (A or B)

(B) Grouping of Logical operators [AND, OR]

else if ADI [A] != ADI [B]
3. if "AND" then (A and B)
4. if "OR" then no grouping takes place

At this level, as each condition is parenthesized, we consider the grouping of conditions and not operands.

(C) Grouping of unary operator: NOT (~)
5. if (condition prefaced by "NOT")
then [~ (Condition)]

(D) Grouping of NAND and NOR

else if ADI [~ A] = ADI [~ B]
6. if "AND" then (~A and ~B) --- [NOR semantics]
7. if "OR" then (~A and ~B) --- [NOR semantics]

else if ADI [~ A] != ADI [~ B]
8. if "AND" then no grouping takes place
9. if "OR" then (~A or ~B) --- [NAND semantics]

(E) Associativity Rule:
10. If the operator is repeated then prefer the left one first

Figure 4.1 Grouping Algorithm

Below we explain the possible semantics, which STRUCT can deal with:

(A) whenever a query is parsed, all the wrappers in the query are grouped, before proceeding to next step. Cases 1 and 2 ensure that the wrappers are given higher preference over the logical-operators.

(B) Once the wrappers are grouped, then the binary logical operators “AND, OR” are targeted. In these logical operators, “AND” is given higher precedence over the “OR” operator. Cases 3 and 4 confirm that the “AND” logical operator is grouped before “OR” logical operator, if both of them occur simultaneously.

(C) The grouping of “NOT” delimiter takes place, once all the conditions in the sub-query are parenthesized. At this stature, it‘s checked if, whether in the sub-query, the condition was prefaced by “NOT” or not. If yes, then the “NOT” is simply prefixed before the condition as: NOT (condition). In case 5, the “NOT” in STRUCT, is treated as a delimiter, and hence the condition formed, is prefaced by “NOT”.

(D) In NAND and NOR semantics, the “NOT” is treated as unary operator and hence case 6 to 9 ensures that “NOT” is given higher priority over the “AND, OR” operators. Whenever user implies “neither…nor (none)” in the query, then such condition is represented by using NOR operator. Case 6 and 7 certifies that the NOR semantics is grouped properly. When user states “either of them but not both” in the query, then such condition is represented by using NAND operator. Case 9 takes care of NAND semantics.

(E) Case 10 helps STRUCT to preserve the Associativity rule. In the user query, if the same operator is repeated, then the precedence is given to the left hand side operator. Here, it‘s checked whether the grouping is possible or not. If not, then the pointer is advanced to the next successive operator and so on. STRUCT also takes into account the following correlative conjunctions: both...and, either...or, neither...nor, not...but, not only...but also. All these above conjunctions are stated in terms of OR,AND, and NOT, before parsing the query.

We detail the above cases by illustrating it with an example.

CASE 1, 2: If the considered operator is "and" & if it operates on the values belonging to the same attribute then replace the “and” by “or”, and group the operands.

Query 15: Find cars with Red and blue Color and 4 doors

Equivalent semantics: Find cars with Red or blue Color and 4 doors

Parenthesized format: (Red or Blue) Color and (4) doors.

Here, the ambiguity of whether the operand “blue” belongs to “or” or “and” is resolved.

CASE 3: If the considered operator is "and", and if it operates on the values belonging to the different attributes then they can be grouped.

Query 16: Find a car which is white in color and price < 2000 or manufactured in 2002

Parenthesized format: [(White (color) and 2000 (price)) or (2002) year].

Here, the ambiguity of whether the operand “2000” belongs to “and” or “or” is resolved.

CASE 5: In this case, the “NOT” act as a delimiter, in order to form a condition prefixed by NOT.

Query 17: Find car which is not of model Honda and red in color

Parenthesized format: not (Honda and Red)

Here, “and” is given higher precedence than “not”, since “not” acts as a delimiter and not a unary operator.

CASE 6, 7: If the considered operator is "and", and if it operates on the values belonging to the same attribute but prefaced with “not”, then they are grouped together.

Query 18: Find a car which is not Blue and not Red in color but has 4 gears.

(OR) Find a car which is not Blue or not Red in color but has 4 gears.

Equivalent NOR semantics: Find a car which is neither Blue nor Red but has 4 gears.

Parenthesized format: (not Red and not blue) and (4) gears

Here, the “not” acts as a unary operator and hence is given higher preference over the “and” operator.

CASE 9: If the considered operator is "or" & if it operates on the values belonging to the different attributes but prefaced with “not”, then they are grouped, as in Query 19.

Query 19: Find a car which is not white in color or not Toyota in model but has 4 gears

Equivalent NAND Semantics: Find a car which is either white or Toyota in model but not both and has 4 gears

Parenthesized format: (not white or not Toyota) and (4) gears

Here as well, the “not” acts as a unary operator and hence is given higher preference over the “or” operator.

CASE 10: If operator is successively repeated, then as per the Associativity rule, left hand side operator is given the first preference. Check whether the grouping takes place or not. If not, go for the next operator. Query 20 is an example.

Query 20: Find a car having color Red or 2 or 4 doors and 5 gears

Parenthesized format: (Red) color or (2 or 4) doors and (5) gears

Here, as the second “or” acts as a wrapper, it is given higher precedence compared to the other logical operators. And hence resolving the ambiguity for the operands “2, 4”. If only one operator is present then no grouping take place as there is no question of precedence. Note, for the simplicity purpose, most of the examples in above cases, have only one sub-query in the predicate. Consider the following multiple sub-queries example, to illustrate the working of grouping algorithm.

Query 21: Find a Honda car with white or black color and 4 doors or having a blue color with 2 doors. Based on the delimiters, the query is divided and further parsed.

Step 1 – Dividing the query:

Subject: Honda car

SQ-1: white or black color and 4 doors or

SQ-2: blue color

SQ-3: 2 doors

Step 2 – Grouping:

A] Operand level grouping

Subject: (car = Honda)

SQ-1: (color = white) or (color = black) and (doors = 4) or

SQ-2: (color = blue)

SQ-3: (doors = 2)

B] Now if there are any wrappers in the above sub-queries, then group them first.

ADI for SQ-2 is: color or color and doors. As, the "or" operates on the same attribute domain, it is treated as wrapper, grouping the (white or black) values. So now we have:

Subject: (car = Honda)

SQ-1: (color = white or color = black) and (doors = 4) or

SQ-2: (color = blue)

SQ-3: (doors = 2)

C] Condition level grouping:

Only in SQ-2, we have multiple conditions. By using CASE 3 of grouping algorithm, we can group those two conditions.

Subject: (car = Honda)

SQ-1: ((color = white or color = black) and doors = 4)) or

SQ-2: (color = blue)

SQ-3: (doors = 2)

D] Sub-query level grouping:

Now, with the help of CASE 4, it‘s clear that SQ-1 and SQ-2 cannot be grouped. But then, by using CASE 3, we can group SQ-2 and SQ-3. Since both of the sub-queries have different attribute domain information (color, doors).

Therefore, after grouping we have:

Subject: (car = Honda)

SQ-1: ((color = white or color = black) and doors = 4) or

SQ-2: ((color = blue) and (doors = 2))

Now by using property 2 of ADI, we can notice that the SQ-1 and SQ-2 both have the attribute information: (color and doors). Also both of the sub-queries are connected by "or". As the ADI‘s are same, "or" is treated as logical operator and hence with the help of CASE 2, we group the SQ-1 and SQ-2, resulting into:

Subject: (car = Honda)

Predicate: (((color = white or color = black) and doors = 4) or ((color = blue) and (doors = 2))) Now, as all the grouping at sub-query level has been done, we have grouped subject and predicate.

E] Query level grouping:

Finally, we associate, the subject with the predicate, resulting into the parenthesized query. Parenthesized Query: ((car = Honda) and (((color = white or color = black) and doors = 4) or ((color = blue) and (doors = 2)))).

The algorithm terminates, as no further grouping is possible. Note that, if the user doesn‘t explicitly specify attribute information for the value, then STRUCT speculates a list of attributes by referring the inverted index (Attribute_info Table 1). So the number of parenthesized formats built for a query will be equal to the multiplication of number of speculated attributes for each such value for which user hasn‘t specified the attribute information. Later in section 5, we will see how this missing information incurs an additional burden on performance of the search engine.

4.5 Link Data-Structure (LDS)

At this stage we have the parenthesized formats of the given English query. LDS data structure is used to represent the above parenthesized query in tabular format. Each parenthesized format has its own associated LDS. Now, for a particular parenthesized format, each value along with its associated information is represented by a row in the table. Part of the associated information, like values, their attributes and comparison operators associating them, is derived from ADS. Further, the list of tables, representing the given attribute for a value, is also noted down. To build LDS, the parenthesized query is parsed from left-to-right, and whenever a value is encountered, an entry is made along with the associated information. For each value, its attribute name (either specified in the query, or taken from the list of speculated attributes), and table names where the attribute appears, is entered. Also if the value (operand) is preceded by the open parentheses then "("entry is added to the Open_bracket column, else if the value is followed by the close parentheses then ")" entry is added for the Closed_bracket column. Further, comparison operators between the value and attribute, and the logical operators between the conditions are also entered. Consider the sample Query: V1 and V2 or V3 Parenthesized Query: (V1 and V2) or V3.

The LDS for the above sample query is as follows:

Table 4.2 LDS

Open	Table	Attribute	C	**Value**	Closed	L
(	T1...Tm	A1	=	V1	-	AND
-	T1...Tn	A2	>	V2	)	OR
-	...	...	...	V3	-	-

Where, "C" – Comparison operator and "L" – Logical operator

4.6 Possible Combinations

The above tabular format has to be further mapped to SQL query. To do so, we need to make the following combinations:

i. Across Row Combination

ii. Unorganized format

iii. Structured format

In LDS, each row represents a condition or a part of it. Now, to formulate an entire query out of LDS, we need to append all the rows. But in the underlying database a particular attribute (e.g. A1), can be associated with multiple tables (T1…Tm), and so can be true for most of the attributes. So we need to compute all sorts of possible combinations across the rows [m*n*…], to enumerate different possible conditions forming a query.

Once we have these combinations, the structure of each such resulting combination is referred to as the *unorganized format* for the considered query. Unorganized format for one of the sample queries is shown in Query 22.

Query 22: Find a Honda car which is Civic in model and mileage greater than 20 or has price less than 15000 or manufactured in year 2000.

SQ-1: Find a Honda car

SQ-2: Civic in model and mileage greater than 20 or SQ-3: price less than 15000 or manufactured in year 2000

For SQ-1, the unorganized format is shown in Table 7.

Table 4.3 Unorganized Format for Query 22

<table>
<tr><td>Row 1</td><td></td></tr>
<tr><td>((| Manufacturer | make | = | Honda |) | -</td><td>+</td></tr>
<tr><td>Row 2</td><td></td></tr>
<tr><td>(| Models | Model_name | = | civic | - | and</td><td>+</td></tr>
<tr><td>Row 3</td><td></td></tr>
<tr><td>-| VehicleDetails | mileage | > | 20 |) | or</td><td>+</td></tr>
<tr><td>Row 4</td><td></td></tr>
<tr><td>(| VehicleDetails | price | < | 15000 | - | or</td><td>+</td></tr>
<tr><td>Row 5</td><td></td></tr>
<tr><td>(| VehicleDetails | year | = | 2000 | - | and</td><td>+</td></tr>
<tr><td>Row 6</td><td></td></tr>
<tr><td>- | VehicleFeature | Air_condition | = | YES | - |</td><td></td></tr>
</table>

[Note: “+” – append, “|” – Separator, “-” – NULL or void]

Overall, the computational time for possible combinations of a given English query, is directly proportional to the number of attributes associated with a particular value, and number of tables associated with a particular attribute. So if the user explicitly specifies the attributes for the corresponding values in the query, then the number of combinations, and hence the computational time can be minimized; otherwise, STRUCT speculates the list of attributes by referring attribute_info table (1) to enumerate the possible parenthesized formats for the query, which in turn incurs additional time complexity overhead for enumerating LDS, and unorganized formats for the query. (Refer Figure 9 for details.)

4.7 Constructing SQL Clauses

Now we have to derive the SQL query from these unorganized formats of the query. The basic cell or element that comprises the unorganized format of a query is shown below.

Table 4.4 Unorganized Format of Cell

Open bracket	Table List	Attribute List	C	**Value**	Closed bracket	L
(1)	(2)	(3)	(4)	(5)	(6)	(7)

This cell basically represents a value along with all the possible information surrounding it. As the unorganized format of each value is already appended (+), it formulates into a condition, then into a sub-query and eventually into a query. Now to create an SQL statement, every cell from the unorganized format of a query, is taken into account. Each cell, adds some information to the SELECT, FROM and WHERE clause. The construction of the clause is as follows:

i. SELECT CLAUSE:

Consider all the distinct Table**(.)Attribute** pairs from all the cells and separate them by ‘,’ delimiter. This forms our select clause. For example, the SELECT clause for query 22 is: “SELECT Manufacture.Make, Models.name, VehicleDetails.mileage, VehicleFeature.price, VehicleFeature.year”

ii. FROM CLAUSE:

Consider all the distinct **Table** names from all the cells. Now it is checked whether there exists a path between all these tables, linked by Primary – Foreign key relationship. If the path exists, then it implies that this table combination can be used to execute the query. For example, the FROM clause for query 22 is: “FROM Manufacture, Models, VehicleDetails, VehicleFeature”

iii. WHERE CLAUSE:

STRUCTS appends the unorganized format of each cell in the follow way: Open bracket + Table(**.**)Attribute + comparison_operator + value + closed_bracket + logical_operator + … + next cell [Replace ‘-’ by blank space, if any]

For example, the WHERE clause for query 22 is: “WHERE ((Manufacture.Make=‘Honda‘) and (Models.name=‘Civic‘ and VehicleDetails.mileage > 20) or (VehicleFeature.price < 15000 or VehicleFeature.year=2000))”

Eventually all the clauses are grouped together to form an SQL statement, which is then executed on the underlying database to retrieve the result.

Chapter 5

Experimentation and Evaluation

For the demonstration purpose we have used the well-known banking relational database; one can refer the book [19] for the data tuples and schema information. In STRUCT, JAVA programming language is used to implement the source code, and MySQL to create the relations; making the system platform independent. Further, JDBC driver is used to connect the interface at frontend with the database in backend.

5.1 Sample Results:

The schema graph for the considered Banking database is shown in Figure-1. Below, we present screen shots (Figure 5 to 7) for some of the sample queries, executed on search engine STRUCT.

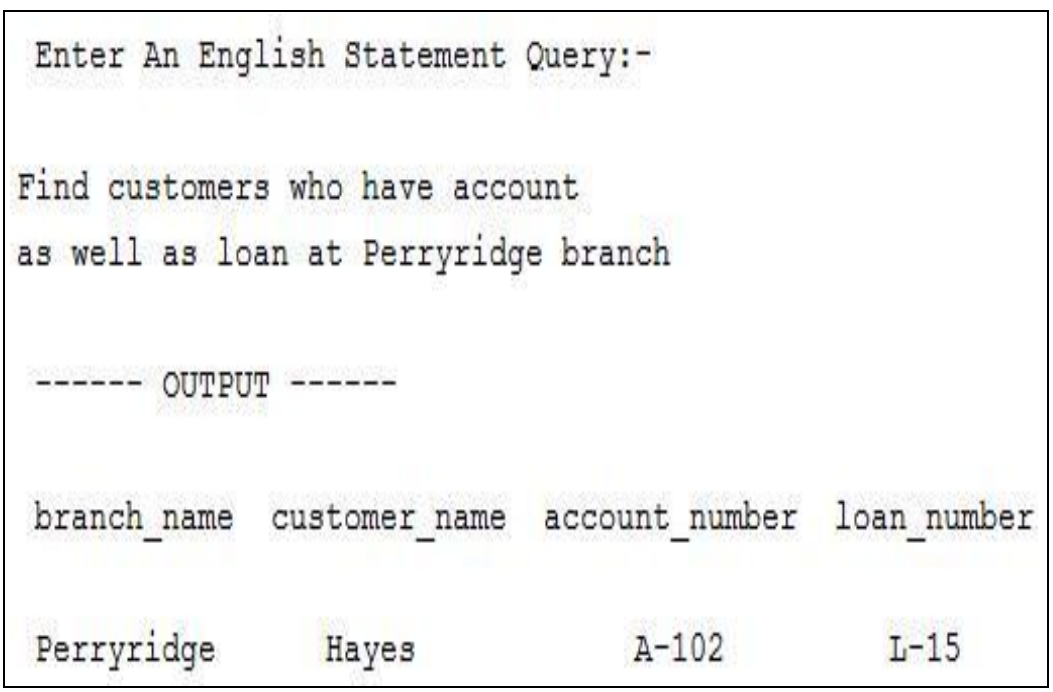

Figure 5.1 Screen Shot for sample Query-23

```
Enter An English Statement Query:-

Look for the customers who do not
have an account at Brighton branch

------ OUTPUT ------

branch_name     customer_name     account_number

  Perryridge        Hayes            A-102
  Downtown        Johnson            A-101
  Redwood         Lindsay            A-222
  Mianus            Smith            A-215
  Round Hill       Turner            A-305
```

Figure 5.2 Screen Shot for sample Query-24

```
Enter An English Statement Query:-

List the customers living in Harrison or Stamford
and having balance less than 500 dollars

------ OUTPUT ------

customer_city    balance    customer_name

  Harrison       400.00       Hayes
  Stamford       350.00       Turner
```

Figure 5.3 Screen Shot for sample Query-25

5.2 Performance Evaluation:

The performance of STRUCT can be evaluated based on how closely the constructed SQL statement matches the meaning of the given English user query. So long as the constructed SQL is semantically matched, the results should be relevant to the user. So to evaluate the correctness of the SQL statement, we have to analyze each clause individually which comprises the SQL statement. Firstly, the SELECT clause deals with displaying the information that user is looking for. STRUCT makes sure to display all the attributes appearing explicitly and implicitly (values for which attribute information is missing) in the user query and hence giving the required information to the user. So the definiteness of the constructed SELECT clause is as good as needed. Secondly, the FROM clause identifies and associates relations through foreign links in order to formulate a join condition. To assess such join conditions in the FROM clause, we need to know whether the paths constructed by these joins are relevant to the user query or not; since a valid paths implies a valid FORM clause. As in the case of information retrieval (IR), such evaluation can be done in terms of Recall and Precision measures. Since ***recall*** is the fraction of the paths that are relevant to the query that are successfully retrieved, STRUCT always comes up with a perfect score since all the eligible or relevant paths, used to answer the query, are constructed. As for ***precision***, it is the fraction of retrieved path that are relevant to the search. STRUCT uses every bit of the information from the user query to build its equivalent SQL query. But sometimes it may happen that, the user provides less information than needed. For example, consider the following two queries: Query A — Find a Green car. Query B — Find a Green color car.

Now to generate FROM clause for Query B, we just need to check for a path that connects the relations having attributes ‘car’, and ‘color’ (which covers the value ‘Green’). But for Query A, as the attribute information for value ‘Green’ is missing, STRUCT may interpret ‘Green’ in a different way; since in the underlying database, the value ‘Green’ can be represented by more than one attribute. In such a case, STRUCT generates a list of speculated attributes by referring the attribute_info table (1), for the value ‘Green’; for example, the list of speculated attributes for ‘Green’ might include lantern, tea, customer name, color, streetlight and much more. Further, to generate a FROM clause for Query A, we need to check whether a path, which connects the relation having attribute ‘car’ with the relation representing the considered speculated attribute, exist or not. So it implies that as the list of speculated attribute grows, the possible combinations of finding paths will manifold. Most of the combinations will simply be discarded, if no path exists between the considered relations; whereas if it exist, a FROM clause is generated out of it. Overall, as compared to Query B, the additional computation needed to generate the invalid combinations results in decline of precision for Query A. Therefore, the more the query is precise, the more is the precision. To understand the impact of attributes specified in the given query, we should consider computational time, recall, and precision measures. In figure 8, we graphically demonstrate the implications of these measures with the attributes encountered in the query.

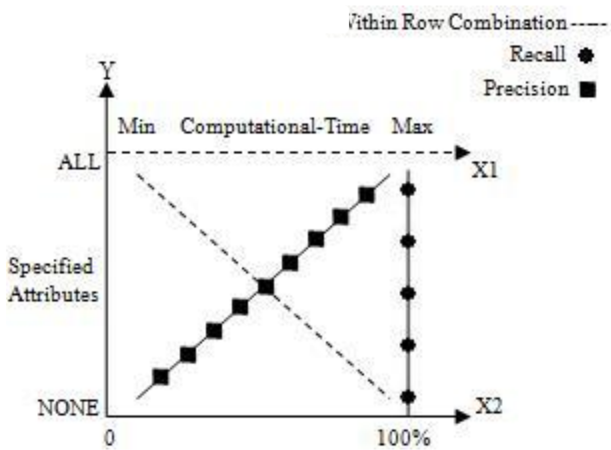

Figure 5.4 Impact of Specified Attributes

In Figure 8, the Y-axis represents number of values in the given query for which the attributes are specified explicitly. X1-axis represents the time factor for query computation; whereas X2-axis denotes the percentage value for recall and precision. So for a query at hand, as the user specifies more and more meta-data (attribute) information the computational time decreases and the precision increases. The recall is constant though, since irrespective of meta-data information (represented by Y-axis), STRUCT eventually generates all the valid paths responsible to answer the user query.

Finally, the WHERE clause represents and constitutes the semantics of the user query. The construction of WHERE clause in STRUCT, is based on how the English query is broken down into sub-queries, with the help of delimiters and then how they are grouped back with the help of attribute domain information in the Grouping Algorithm. Therefore, the performance of the WHERE clause is determined by how well the user query is parenthesized to justify the operator precedence, in order to retain the semantics of the given query (There are no agreed upon benchmarks for evaluating the semantics of the given query). By far, our experiments and results shows that the semantics of the user query is retained in its entirety, in the constructed SQL-WHERE clauses.

Overall in STRUCT, the given user query is mapped to its equivalent SQL statement by taking the query connotation into consideration. The correct results are the answers returned by the corresponding schema-aware language, namely SQL. But if the user in his/her query doesn‘t specify contextual information (in terms of metadata, operators, delimiters, conjunctions as listed by the STRUCT‘s CFG grammar) and sticks to keyword queries only, the system still works, as STRUCT collapses/degenerates to conventional Keyword Search System (KWS). In this approach, as query consists of keywords only, the system attempts to find and connect the relations, through primary-foreign links, having the query keywords (AND semantics only). It further constructs an equivalent SQL statement out of it, to retrieve the results. But it does not go for the partially matched results (representing OR semantics), as that distorts the semantics of the given query; the very motive of STRUCT.

Chapter 6

RELATED WORK

As indicated before, most of the related work in the field of "search on structured database" is restricted to handling of queries with keyword only; and hence curbing the liberty of user from exploiting the potency of natural language queries. Below we choose several of them to emphasize the limitations of the existing search systems. The interface provided by most of the prototype systems [1,2,3,7,9,10,11,12,13,14,15,16,17,18,20] accept only keywords (data and meta-data information) from the user; any additional information, if provided, is not taken into account. The drawback of such keyword search is that the query can be interpreted variously and hence large numbers of ambiguous answers are returned; which further creates a need of ranking the results as per the relevance. Although several authors, reference [8], noted the need of query interpretation, none of them offered practical ways of dealing with them. In contrast, STRUCT allows user to construct English statement queries by giving them the complete charge of using natural language format, and not just limiting them to the data and meta-data nomenclature. Furthermore, as most of the search systems restrict themselves to the keyword queries, they are indebted to the use of either conjunctive or disjunctive semantics. Also in these search engines, the user is not allowed to specify the semantics or requirements explicitly; instead, the engines make the assumptions and run the query initially with AND semantics and then with OR semantics to rank the equivocal results. For example, DBXplorer[2], BLINKS[10], DISCOVER[11], BANKS[3], and EKSO[20] are limited to AND semantics only. DISCOVER2[12], [7], [16], SPARK[17] and

KITE[18] does go one step ahead to include AND, OR semantics. But the user cannot use these semantics simultaneously in his query; since either AND or OR is taken individually into account and not both. Reference [9] does address AND, OR and NOT semantics, but then here it requires that, for each keyword the user has to explicitly specify the level of search, which may be table, column, or tuple; making it quite cumbersome for the user. Whereas, STRUCT does allow simultaneous use of operators, including "AND, OR, NOT", giving user the autonomy of expressing their semantics. Additionally, it also considers the comparison operators and correlative conjunctions, if any.

Ranking technique in this field, usually has two well known approaches. DBXplorer[2] and DISCOVER[11], ranks the results based on number of relational joins involved in the computation; smaller the distance between two tuple units, the more relevance exist between them. BANKS[1,3,13] use node weights and edge weights to determine the relevancy of result. Whereas, DISCOVER2[12], SPARK[17], and [16] use IR style Ranking function to rank the queries, taking TF-IDF frequencies into account. In these systems the IR style approach works perfectly fine to rank the completely or partially matched results. But as these systems does not allow user to express the query semantics, they don‘t know the exact answer and consider all possible set of answers. And hence the results represent a mixture of different query semantics, where any combination of keyword occurrences is found and returned to the user. In STRUCT, similar to DBXplorer[2] and DISCOVER[11], the results are ranked based on number of relational joins, but then it also takes into account the semantics of the user query, making the result more meaningful exact, and accurate to the requirement.

Various alternatives have been followed to make the database representations look simpler. BANKS[1,3,13], BLINKS[10], and EASE[14] represents the database by using data graph. Here, as tuples are represented as nodes, insertion of new records will lead to addition of nodes as well of their corresponding edges, and hence resulting in more memory space consumption as the graph grows. Reference [15] models tuple unit (materialized tuples) as a node in the graph, and hence the resulting graph has much smaller size than the data graphs. But it incurs additional pre-computation of joining the associated relations in the database to enumerate all the tuple units, and further possibly integrating them. STRUCT, similar to most of the other systems[2,11,12,16,17] does represent the database by using schema graph, which is significantly smaller than the data graph.

Chapter 7

CONCLUSION AND FUTURE WORK

The main objective of STRUCT is to free end-users from considerations related to the structure and association of the underlying relational database. STRUCT provides an interface where users can specify queries having conjunctive, disjunctive and negative semantics. Further, the user is not restricted to the terminologies used in the underlying database, as STRUCT refers thesauri to understand the user‘s intended terms. Additionally, STRUCT gives user the liberty to submit query having multiple sub-queries. STRUCT further helps the novice user get rid from the additional burden of learning Structured Query Language (SQL), used to query the structured database. Instead, the user can simply use the English language statements to retrieve the desired results. By employing a relatively simple parsing technique and developing a grouping algorithm which incorporates contextual information obtained from user queries, STRUCT is able to form SQL queries reflecting users‘ intention. The user English query is mapped to its equivalent SQL statement in the background, hiding the complications from the end-user. To optimize the performance, query having less number of join relations, is picked up for the execution purpose.

In summary, by incorporating contextual information contained in user queries, STRUCT goes beyond many other database keyword search systems can offer. Yet, many further improvements are still needed. This includes supporting phrase search, creation of a complete list of possible delimiters to make the search engine more effective, incorporating a readymade thesaurus to increase the efficiency, and inclusion of ontology to solve queries where the user requirement is generalization of many specific instances in the database. For the time being, the thesauri used by STRUCT, is managed by the administrator and hence is a manual and time consuming process of updating it. We are currently extending the STRUCT system to handle query search on XML data as well. Also our future goals would be to consider the clauses, operators or functions that can help user to specify their needs and requirements with much ease and convenience. So far, we have implemented the main three clauses (select, from and where) of the select statement. We would like to extend this work by considering the GROUP BY and HAVING clause used to implement the aggregate functions. Most frequently used grouping functions in the natural language are Average, Sum, Min, Max and Count. Additional operators like ANY, SOME, BETWEEN and ALL are also used extensively in the natural language. So we would further like to consider the feasibility of implementing and incorporating the above features in our search engine, named STRUCT.

REFERENCES

[1] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, C. Nakhe, Parag,and S. Sudarshan. Banks: Browsing and keyword searching in relational databases. VLDB 2002. [2] S Agrawal, S Chaudhuri, G Das: DBXplorer: A system for keyword-based search over relational databases. ICDE 2002.

[3]G. Bhalotia, A. Hulgeri, C. Nakhey, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. ICDE 2002.

[4] Y. Chen, W. Wang, Z. Liu, and X. Lin. Keyword Search on Structured and Semi-structured Data. SIGMOD, 2009.

[5] Y. Chen, W. Wang, Z. Liu. Keyword-based search and exploration on databases. ICDE 2011 Tutorial

[6] J. Coffman, A. C. Weaver. A framework for evaluating database keyword search strategies. CIKM, 2010.

[7] K. V.V Ganeshan, N.L. Sarda, S. Gupta. Keyword search in geospatial database. ACM GIS, 2010

[8] D. Haam, K. Y. Lee, M. H. Kim. Keyword search on relational databases using keyword query interpretation. Proc. Int'l Conf. Computer Sciences and Convergence Info. Tech (ICCIT 2010).

[9] M. Hassan, R. Alhajj, M. J. Ridley, K. Barker: Database selection and keyword search of structured databases - Powerful search for naive users. IEEE, 2003.

[10] H. He, H. Wang, J. Yang, and P. Yu. Blinks: Ranked keyword searches on graphs. SIGMOD, 2007.

[11] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. VLDB 2002.

[12] V. Hristidis, L. Gravano, Y. Papakonstantinou:Efficient IR-Style Keyword Search over Relational Databases.VLDB 2003.

[13] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. VLDB, 2005.

[14] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-
structured and structured data. SIGMOD, 2008.

[15] G. Li, J. Feng, J. Wang. Structure-aware indexing for keyword search in databases. CIKM 2009.

[16] F. Liu, C. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. SIGMOD, 2006.

[17] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: Top-k keyword query in relational databases. SIGMOD, 2007. [18] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient keyword search across heterogeneous relational
databases.ICDE, 2007. [19] A. Silberschatz, H. F. Korth, S. Sudarshan. Database System concepts. 5th Edition, 2006. [20] Q. Su and J. Widom. Indexing relational database content offline for efficient keyword-based search. IDEAS, 2005.

[21] J. X. Yu, L. Qin, L. Chang. Keyword search in relational
databases: A survey. ICDE 2010