Distribution Agreement

In presenting this thesis as a partial fulfillment of the requirements for a degree from Emory University, I hereby grant to Emory University and its agents the non-exclusive license to archive, make accessible, and display my thesis in whole or in part in all forms of media, now or hereafter now, including display on the World Wide Web. I understand that I may select some access restrictions as part of the online submission of this thesis. I retain all ownership rights to the copyright of the thesis. I also retain the right to use in future works (such as articles or books) all or part of this thesis.

Yicong Li April 3, 2018

The Translation from SQL to Relation Algebra

by

Yicong Li

Shun Yan Cheung

Adviser

Department of Mathematics and Computer Science

Shun Yan Cheung

Adviser

Jinho Choi

Committee Member

Juliette Stapanian Apkarian

Committee Member

2018

The Translation from SQL to Relation Algebra

By

Yicong Li

Shun Yan Cheung

Adviser

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Sciences with Honors

Department of Mathematics and Computer Science

2018

Abstract

The Translation from SQL to Relation Algebra

Yicong Li

SQL (Structural Query Language) and Relational Algebra are two important languages to manipulate relational database. SQL is an international standard language used to express queries on data stored in a database. Relational Algebra is a Mathematical language with operations on sets. SQL queries are first translated to an equivalent expression in Relational Algebra in query processing. The thesis explores the translation from SQL to Relational Algebra to gain a deeper understanding in database systems. The thesis begins with an introduction to the problem (including motivation to working on the translation), the related background knowledge to handle the translation, and follows with the project design. It then discusses the evaluation of the result, reflects on my learning experience from the project, and makes suggestion about further improvement.

The Translation from SQL to Relation Algebra

By

Yicong Li

Shun Yan Cheung

Adviser

An abstract of
a thesis submitted to the Faculty of Emory College of Arts and Sciences
of Emory University in partial fulfillment
of the requirements of the degree of
Bachelor of Sciences with Honors

Department of Mathematics and Computer Science

2018

Table of Contents

CHAPTER PAGE

CHAPTER 1 – Introduction **1**
1.1 Motivation 1
1.2 Problem Description 2
1.3 Related Work 3

CHAPTER 2 – Background **5**
2.1 Relational Database 5
2.2 Relational Algebra 6
2.3 SQL 8
2.4 Parsing 10
2.4.1 Lexical Analysis 10
2.4.2 Regular Expression 11

CHAPTER 3 – Project Description **14**
3.1 Introduction 14
3.2 The Lexical Analyzer of the Project 14
3.3 Parsing the SELECT Clause 16
3.4 Parsing the FROM Clause 18
3.5 Checking and Filling the Attribute List 20
3.6 Parsing the WHERE Clause 20
3.7 Converting to Relational Algebra 29

CHAPTER 4 – Test and Evaluation **31**
4.1 Sample Outputs 31
4.2 Limitations 36

CHAPTER 5 – Project Evaluation and Future Work **37**
5.1 Project Goal and Assessment 37
5.2 Improvement 38

List of Tables

Table 2.1 Relational Algebra Model....6
Table 2.2 Relational Table 1....7
Table 2.3 Relational Table 2....7
Table 2.4 Relational Table 3....8
Table 2.5 Relational Table 4....8
Table 3.1 Lex Definitions....15
Table 3.2 Token Codes....15
Table 3.3 SQL Keywords....16
Table 3.4 Parsing Attribute List....18
Table 3.5 Parsing Relation List....20
Table 3.6 Grammar of WHERE Clause....21

List of Figures

Figure 1.1 Process of Query Compiler 2
Figure 2.1 Finite Automaton Diagram 1 12
Figure 2.2 Finite Automaton Diagram 2 13
Figure 3.1 Structure of AttrType 17
Figure 3.2 Structure of RelaType 19
Figure 3.3 Parsing Tree 1 22
Figure 3.4 Structure of Node 23
Figure 3.5 Parsing B-expression 1 24
Figure 3.6 Parsing B-expression 2 24
Figure 3.7 Parsing B-factor 1 26
Figure 3.8 Parsing B-factor 2 26
Figure 3.9 Parsing Expression 27
Figure 3.10 Parsing Expression Example 28
Figure 3.11 Relational Algebra Tree 29
Figure 3.12 Structure of Relational Algebra Node 30
Figure 4.1 Output 1 31
Figure 4.2 Relational Algebra Tree 32
Figure 4.3 Output 2 32
Figure 4.4 Output 3 33-34
Figure 4.5 Output 4 35
Figure 5.1 Query Object that Contains a Correlated Sub-query 39
Figure 5.2 Relational Algebra Tree Manipulation 1 40
Figure 5.3 Relational Algebra Tree Manipulation 2 41
Figure 5.4 Query Object that Contains Attribute IN (Sub-query) 42
Figure 5.5 Relational Algebra Auxiliary Tree Example 42
Figure 5.6 Relational Algebra Tree Manipulation Example 43

Chapter 1

Introduction

1.1 Motivation

The explosive growth of data has posed challenges as well as provided opportunities for us. Structured information, such as personnel records, are commonly stored and managed by relational database systems. Relational Database Systems store information as relations. A relation is a table containing similar data items, such as employee records. As computer scientists, it is important for us to increase our abilities to manipulate database systems and retrieve information from them efficiently. At Emory University, the CS 377 and CS 554 computer science courses expose students to comprehensive and systematic knowledge on database systems. Students learn how to retrieve information from a database through SQL queries and learn various ways to access the database server, such as through a web-based interface.

Query languages are used to manipulate and retrieve data from a database. There are different languages to manipulate the information stored in relational databases, for example, SQL and Relational Algebra. Structured Query Language (SQL) is a standard and well-known programming language for manipulating relational databases. Relational Algebra is a mathematical language that expresses operations on relational databases. While SQL is application-oriented and allows the users to express “what information” they want to retrieve with conditions, Relational Algebra is more procedural and express “how” the data is retrieved using relational operations. Therefore, in order to process an SQL query, the first step is to

translate the query into an equivalent Relational Algebra expression. Thus, Relational Algebra serves as a medium that moves users from expressing what they want from the database to executing a set of instructions/operations to obtain the data. In fact, the translation from SQL to Relational Algebra is the initial step in the query execution process. The query execution plan expressed in Relational Algebra can be further optimized. Therefore, I am inspired to study the translation process in detail by constructing a program to perform the translation.

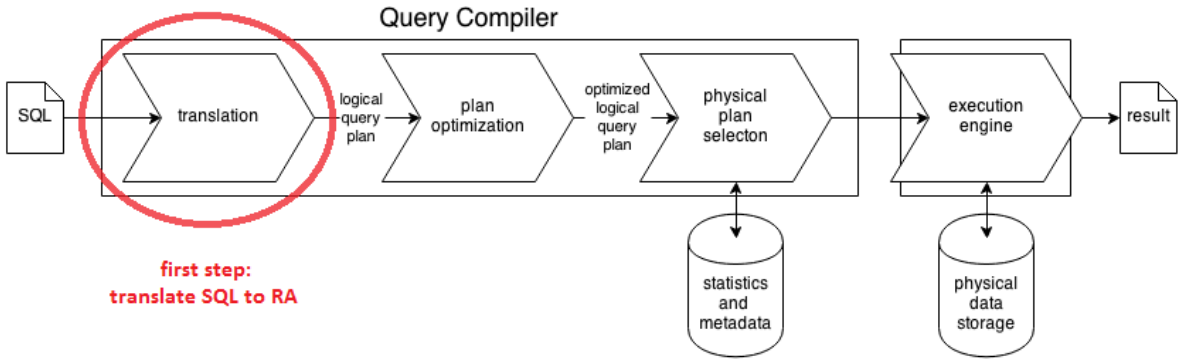


Figure 1.1: Process of Query Compiler

1.2 Problem Description

In this thesis, we study the process of translating SQL queries into Relational Algebra expressions. SQL queries are in fact "sentences" in a specified language: the Structured Query Language. The process to divide a language into small components that can be analyzed is called *parsing*.

In order to translate SQL queries to Relation Algebra, we need to parse SQL queries. It includes checking the SQL statement for syntactic and semantic validity and validating each

language element of the statement according to the grammar rules of the language. In addition, the SQL parser uses information about the database content to assign meaning to names used in the SQL query. This information is called “metadata”. After validating the components of a query with the metadata (data that describes the data stored in a database), the SQL parser uses appropriate data structures to store the information obtained from parsing. We need to find out which relations (tables of tuples that contain related attributes) in the database to access. We also need to identify whether the relations or the data fields are renamed to avoid ambiguity. We must also extract information about the qualification regarding the tuples in the relations. All the information about the SQL query is then stored in a query object variable. Finally, the information in the query object is transformed into a sequence of Relational Algebra operations.

The choice to work on relational database originates from the efficiency of using relational models to manage data. A relational database represents data in tables and rows and allows the information of different tables to link together (through the use of keys).

1.3 Related Work

The idea of constructing a program to translate from SQL into Relational Algebra has been realized before. It refers to the paper of “Translating SQL into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries” by Stefano Ceri and Georg Gottlob published on IEEE Transactions on Software Engineering in April 1985. The translator from a subset of SQL into Relational Algebra presented in the thesis is aimed at facilitating the understanding of database systems and practicing implementing parsing through program.

This thesis presents an expository project in the design and implementation of a SQL parser that translates a subset of the SQL language to Relational Algebra. The thesis is organized as follows: In Chapter 2, we discuss the necessary knowledge background knowledge to approach the problem. In Chapter 3, we describe the SQL to Relational Algebra translation including the algorithms and the data structures used. Chapter 4 presents some sample outputs of the parser/translator. Chapter 5 assesses the goal of the project and discusses further improvement on the project.

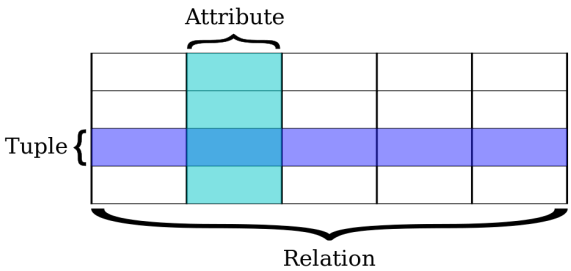
Chapter 2

Background

2.1 Relational Database

A *relational database* is a database based on the relational model of data. The relational model represents the database as collection of relations [1]. Every relation is a table of values in which every row (also known as a *tuple*) is a collection of related data values. A tuple stores information of some object in the real world. A relation schema consists of a relation name and a list of attributes. An attribute is a property of an object. Attributes are the database fields in a relation and are presented as a set of columns in a relation tables.

Figure 2.1 shows an example of two relations. The employee relation has seven attributes: SSN, Name, sex, salary, dno (department number) and bdate (birth date). The department relation contains four attributes: dnumber (department number), department name (dname), manager ssn (mgrssn) and manager start date (mgrstartdate). The sample content of the relations in Figure 2.1 shows how information of 8 employees and 3 departments are stored in the database. Specifically, the employee 'John Smith' works in the department 'Research' because this department has department number 5, which is the department number (dno) of 'John Smith'.



Interpreting Relational Database Model

Employee relation:

SSN	name	sex	salary	dno	bdate

Department relation:

dnumber	dname	mgrssn	mgrstartdate

sample content of employee relation:

SSN	name	sex	salary	dno	bdate
123456789	John Smith	M	30000	5	09-Jan-55
33445555	Frankl Wong	M	40000	5	08-Dec-45
999887777	Alicia Zelaya	F	25000	4	19-Jul-58
987654321	Jennifer Wallace	F	43000	4	20-Jun-31
666884444	Ramesh Narayan	M	38000	5	15-Sep-52
453453453	Joyce English	F	25000	5	31-Jul-62
987987987	Ahmad Jabbar	M	25000	4	29-Mar-59
888665555	James Borg	M	55000	1	10-Nov-27

sample content of department relation:

dname	dnumber	mgrssn	mgrstartdate
Research	5	333445555	22-May-78
Administration	4	987654321	01-Jan-85
Headquarters	1	888665555	19-Jun-71

Table 2.1: Relational Database Model

2.2 Relational Algebra

Relational Algebra is a Mathematical Language that defines operations to manipulate relations [1]. Each operator takes relations as input and yields instances of relations as output through set manipulations. The operations of Relational Algebra can be divided into 3 categories: set operators, relational database specific operators, and aggregate functions.

The most commonly used Relational Algebra operators are:

- $\boldsymbol{\sigma_{condition}(R)}$**:** retrieve tuples(records) in relation R that satisfies the given condition
- $\boldsymbol{\pi_{a_1,a_2\ldots}(R)}$**:** obtain only the attributes a_1, a_2 ... of relation R
- $\boldsymbol{R_1 \text{ X } R_2}$**:** combine the relation R_1 and R_2 into one relation
- $\boldsymbol{R_1 \bowtie_{condition} R_2}$**:** combine the tuples in R_1 and R_2 that satisfy the given condition

  $= \boldsymbol{\sigma_{condition}(R_1 \text{X } R_2)}$

For example, the following Relational Algebra expression will retrieve the name and salary of all employees:

$\pi_{name,\ salary}\ (employee)$

name	salary
John Smith	30000
Frankl Wong	40000
Alicia Zelaya	25000
Jennifer Wallace	43000
Ramesh Narayan	38000
Joyce English	25000
Ahmad Jabbar	25000
James Borg	55000

Table 2.2: Relation Table 1

If you only want to retrieve name and salary of employee who earn more than $30,000， you would use:

$\pi_{name,\ salary}\ (\sigma_{salary>\ 30{,}000}\ (employee))$

name	salary
Frankl Wong	40000
Jennifer Wallace	43000
Ramesh Narayan	38000
James Borg	55000

Table 2.3: Relation Table 2

If you want to retrieve the name and salary of employee who works in the Research department, you would use:

$$\pi_{name,\ salary}\ (\sigma_{dname='Research'}\ employee\ \bowtie_{dno=dnumber}\ department)$$

name	salary
John Smith	30000
Frankl Wong	40000
Ramesh Narayan	38000
Joyce English	25000

Table 2.4: Relation Table 3

If you want to retrieve the name of employees in the Administration department, you can use:

$$\boldsymbol{\pi}_{name}(\sigma_{dname='Administration'\ \wedge\ dno=dnumber}\ (employee \times department))$$

name
Alicia Zelaya
Jennifer Wallace
Ahmad Jabbar

Table 2.5: Relation Table 4

2.3 **SQL**

SQL is an international standard and the most commonly used programming language for manipulating and retrieving data stored in relational databases [1]. The SELECT command in SQL is used to retrieve information from a database. The SELECT command contains clauses to specify the source relations and conditions on the data retrieved. The FROM clause specifies the

relations where the information is to be retrieved from and the WHERE clause provides the condition.

For example:

SELECT name, salary

FROM employee

WHERE salary > 30000

will return name and salary of employees who earn more than $30,000. The corresponding Relational Algebra expression is $\pi_{name,\ salary}\ (\sigma_{salary>\ 30{,}000}\ (employee))$.

To find the name and salary of employee in the "Research" department, we use the following SQL command:

SELECT name, salary

FROM employee, department

WHERE dno = dnumber

AND dname ='Research'

This is equivalent to the following Relational Algebra expression:

$$\pi_{name,\ salary}\ (\sigma_{dname='Research'}\ employee\ \bowtie_{dno=dnumber}\ department)$$

If we only want to find employees in the "Research" department that earn more than $30,000, we would use:

SELECT name

FROM employee, department

WHERE dno = dnumber

AND dname = 'Research'

AND salary > 30,000

which is equivalent to this Relational Algebra expression:

$$\pi_{name}(\sigma_{dname='Research' \wedge salary>30000 \wedge dno=dnumber}\ employee \times department)$$

2.4 **Parsing**

Parsing is the analysis of an input string of the given language by dividing it into different parts according to the rules of the grammar. Parsing consists of two parts: Lexical Analysis and Syntax Analysis.

2.4.1 **Lexical Analysis**

Lexical analysis is the first stage of parsing. It is the process of taking an input string of characters (such as the source code of a computer program) and converting it into a sequence of *tokens*, which have specific meanings in the language. Example of tokens in the English language are verbs, nouns, adjectives, etc. The Lexical Analyzer is a program that performs lexical analysis. The input of a Lex Analyzer is some text, and the output is a stream of tokens. For example, the following input string:

SELECT name FROM employee

will be divided into the following tokens

keyword identifier keyword identifier

2.4.2 **Regular Expression**

The theoretical foundations for Lexical Analysis are Regular Expressions and Automata Theory [3]. Regular Expressions are used to generate pattern of strings. Operators used in regular expressions are:

- **Union(|)** RegExpr1 | RegExpr2
- **Concatenation(•)** RegExpr1•RegExpr2
- **Kleene Closure(*)** RegExpr*

The expression (RegExpr*) represents zero (0) or more occurrence of the given regular expression RegExpr.

Example: suppose **LETTER** = {a,b,c…,z,A,B,…, Z} and **DIGIT** = {0,1,2…,9}

The regular expression:

- **LETTER | DIGIT** = any letter or digit

  such as a, b, A, B, 0,1….
- **LETTER•DIGIT** = any letter followed by any digit

  such as a0, b1, c0….
- **LETTER*** = zero or more letters

  such as ε(empty string), a, aa, aaa, aab…..
- **LETTER• (LETTER | DIGIT)*** = any letter followed by zero or more letters or digits

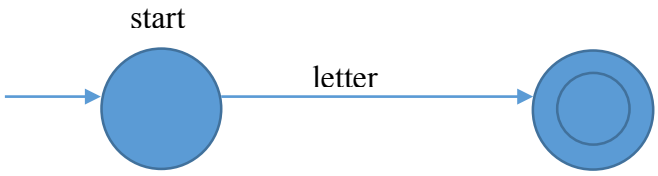
  such as x, xyz, x123

The last regular expression is used to define an identifier in some programming languages.

A finite automaton (FA) is a simple machine used to recognize patterns where the input is a string of characters from a given alphabet. A finite automaton consists of:

- a finite of set S of N states
- a special “start” state
- a set of final (= accepting) states
- a set of transitions from one state to another state (each transition is labeled with characters of the alphabet)

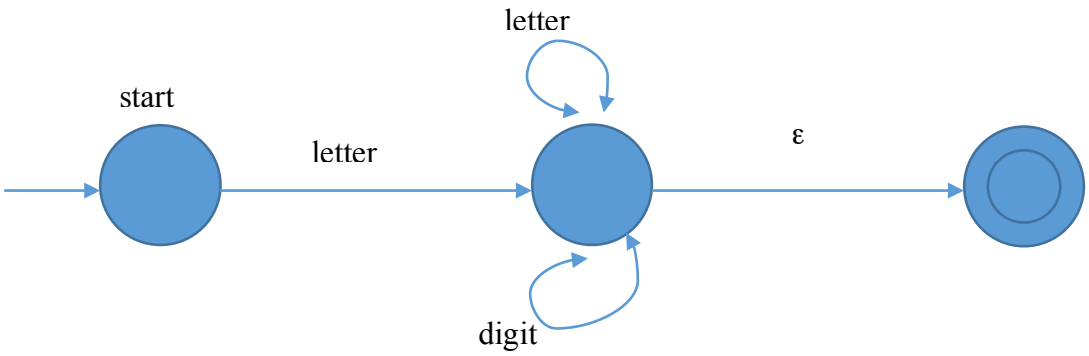
When an input string allows the FA to go from the start state to a finite state, the input string is “valid” or “accepted”. Otherwise, the input string is invalid. For example, the following FA will accept input strings that consist of one letter:



(final state in a FA are denoted by double circles)

Figure 2.1: Finite Automaton Diagram 1

The FA in Figure 2.2 will accept input strings that starts with a letter followed by zero or more letters or digits



(The symbol ε denotes the empty string)

Figure 2.2: Finite Automaton Diagram 2

Chapter 3

Project Description

3.1 Introduction

The objective of the thesis project is to develop a parser for a subset of the SQL programming language and translate input SQL SELECT commands into Relational Algebra expressions. The SQL SELECT query that is implemented in the project consists of the following form:

SELECT attribute-list

FROM relation-list

WHERE condition

where the **condition** can be **(1)** a traditional Boolean condition similar to Boolean expressions in procedural programming languages or **(2)** the SQL "attribute IN (sub-query)" clause.

The SQL parser/translator is implemented in the C-programming language. We used a Lexical Analyzer generator program (Lex) to generate a simple Lexical Analyzer for the project. The main focus of the project was to develop the SQL parser/translator in C. The parsing procedure used in the project is the recursive descent algorithm [3].

3.2 The Lexical Analyzer of the project

In the project, we used the Lex (Lexical Analyzer generator) to produce the Lexical analyzer for the parser. The Lex Lexical Analyzer generator takes as input a file containing

regular expressions (patterns) and generate a function (*yylex*()) that implement the finite state automaton to recognize the patterns. For example, the regular expression "{digit}+" represents a pattern that consists of one or more digits; this regular expression represents an integer in the input. Each token is assigned a unique token code. For example, we assigned the token code 129 to represent an integer token. The regular expression {digit}*+\.{digit}+e+^+{digit}+ represents a floating point number. Example of such a number is $12.5e^{10}$. The Tables 3.1 and 3.2 contain the definitions of the token classes and some of the token codes used in the program.

digit	[0-9]
letter	[A-Za-z]
alpha	({digit}|{letter})

Table 3.1: Lex Definitions

{letter}{alpha}*	128	identifier or keyword
{digit}+	129	integer
{digit}*+\.{digit}+e+^+{digit}+	130	floating number
{digit}+\.{digit}+	130	fixed point number
\'+{alpha}*+\'	131	string
"<="	132	relation operator
">="	133	relation operator
"!="	134	relation operator

Table 3.2: Token Codes

When an identifier is found, we compare it against a table of keywords to check if the identifier is a keyword of SQL. The list of SQL keywords is given in the Table 3.3. When the identifier is equal to one of the keywords, we return the corresponding token code for the keyword. Otherwise, the token code 128 (for identifier) is returned.

```
"ALL", "AND", "ANY", "AS", "AVG",
"BETWEEN", "BY", "CHAR", "CHECK","CLOSE",
"COMMIT","COUNT","CREATE","DECIMAL","DELETE",
"DISTINCT","DOUBLE","DROP","EXISTS","FLOAT",
"FROM", "GO","GROUP","HAVING","IN",
"INSERT","INT","INTO","IS","LIKE",
"MAX","MIN","NOT","NULL0", "NUMERIC",
"OF","ON","OR","ORDER","PRIMARY",
"REAL","SCHEMA","SELECT","SET","SOME",
"SUM","TABLE","TO","UNION","UNIQUE",
"UPDATE","USER","VALUES","VIEW","WHERE",
"WITH"
```

Table 3.3: SQL Keywords

(code values increments from 135)

3.3 Parsing the SELECT clause

The syntax of the first part of the SELECT command is:

SELECT attribute-list

An attribute list is a list of attribute names that are separated by commas. For each attribute name in the list, it can be *renamed* (i.e., aliased) and/or *qualified* with the name of a relation table. Qualifying is prefixing an attribute name with its relation name. When different relations have an attribute with the *same* name, qualifying is necessary to resolve the ambiguity. Therefore, an attribute name in the SELECT list can be specified in 4 different ways:

- A: an *unqualified* attribute name (example: name)
- R.A: attribute name that is *qualified* by its relation name (example: employee.name)
- A L: attribute name that has been renamed (*aliased*) to L (example: name FirstName)
- R.A L: qualified attribute that has been renamed (example: employee.name FirstName)

(R represents the relation name, A represents attribute name and L stands for alias)

We store the parsed attribute names in an array of the type *AttrType*:

The structure *AttrType* constains 3 fields:

```
struct AttrType {

   char attrName[50];   --→ attribute name
   char alias[50];      --→ alias for the attribute
   char relaName[50];   --→ relation name
};
```

Figure 3.1: Structure of *AttrType*

The alias variable contains the alias of the attribute if specified, otherwise, its value is equal to the *attrName* variable.

We use a simple while-loop to parse the list of attributes in the SELECT list. Inside the while-loop, we first parse an identifier. This can the first *unqualified* attribute name or a *relation name* in a qualified attribute name. If the next token is a "." (period), then the identifier found is a relation name and we need to parse the attribute name in the input. Otherwise, the identifier found is an attribute name and we need to fill in its relation name later – when we process the

FROM clause, because an unqualified attribute must belong to some relation in the FROM relations list. Following the code to parse the attribute, we check if the next token is

(1) , (a comma)- in this case, we repeat the loop to parse another attribute.

(2) identifier – in this case, an attribute is renamed to the *alias* given by the identifier

(3) default case: we proceed to parse the FROM clause of the SELECT command.

We store the relation name, attribute name and alias in the SELECT list in the *AttrType* array. For example, the information about the SELECT list in the following SELECT command:

SELECT employee. ssn, sex, name L

FROM ….

is stored in the *AttrType* array as given in Table 3.4

	Attribute 1	Attribute 2	Attribute 3
Attribute name	SSN	sex	name
Alias	SSN	sex	L
Relation name	employee		

Table 3.4: Parsing Attribute List

The relation name of the attributes sex and name are unknown and will be filled in later when the FROM clause is processed.

3.4 **Parsing the FROM clause**

The syntax of the FROM clause in the SELECT command is:

FROM relation-list

The grammar for relation list is a list of relation names that are separated by commas. We use an array of structure *RelaType* to store the information in the FROM clause. Each relation name in the relation list can be in one of the following two forms:

- R: a relation name R (example: employee)
- R B: a relation name R that has be renamed to B (example: employee E)

(B is interpreted as the alias assigned to table R)

The structure *RelaType* is as follows:

```
struct RelaType {
    char relaName[50];   --→ relation name
    char alias[50];      --→ alias
};
```

Figure 3.2: Structure of *RelaType*

We also use a simple while-loop to parse the list of relations in the FROM clause. The list of relation names is stored in the *RelaType* array called *relalist*. Inside the while-loop, we first parse an identifier and store it in the first *relalist* array element. Then the while loop will take one of the following three steps based on the next token in the input:

(1) , *(a comma)* - in this case, we repeat the while loop to parse another relation name

(2) *identifier* – in this case, the identifier is an alias

(3) *default case:* we proceed to parse the (optional) WHERE clause of the SELECT command

For example, the information about the FROM clause in the following SELECT command:

SELECT employee. SSN, sex, name L

FROM employee, department D

is stored in the *RelaType* array as given in Table 3.5

	Relation 1	Relation 2
Alias	employee	D
Relation Name	employee	department

Table 3.5: Parsing Relation List

Notice that if no alias is given, the alias of the relation is equal to itself. In fact, the alias variable will uniquely identify a relation.

3.5 Checking and filling the attribute list

After parsing the SELECT clause and FROM clause, we need to find the relation in the relation list for each attribute in the attribute list whose relation name is empty. We do so by searching the Meta Data. If *exactly* one relation in the FROM clause contains the attribute name, the corresponding relation is filled in the *attrlist* field, otherwise, the parser will return an error. However, if more than one relation in the FROM clause contains the attribute name, the name is ambiguous and the parser will return an error.

3.6 Parsing the WHERE clause

The SQL WHERE clause specifies the condition that tuples must satisfy to be selected. It has many different formats:

(1) Boolean -expression

(2) attribute IN (SQL-query)

(3) attribute rel-op ALL (SQL-query)

(4) attribute rel-op ANY (SQL-query)

(5) EXISTS (SQL -query)

(6) IS NULL

(7) attribute LIKE “pattern”

Format (6) and (7) are relatively easy to process and formats (2), (3), (4), and (5) are very similar. Due to time constraints, we have restricted to parsing formats (1) and (2).

The grammar used in the project to define the tuple condition in the WHERE clause is given in the Table 3.6.

```
<b-expression>  ::=  <b-term> [<OR> <b-term>]*
<b-term>        ::=  <not-factor> [AND <not-factor>]*
<not-factor>    ::=  [NOT] <b-factor>
<b-factor>      ::=   <b-variable> | <expression> <RELOP> <expression>
                      | attribute IN (query)
<expression>    ::=  <term> [<ADD> <term>]*
<term>          ::=  < factor> [<MULTIPLY> <factor>]*
<factor>        ::= <number> | <string constant> | <identifier> | (<b-expression>)
                    |<ADD><factor>
```

Table 3.6: Grammar of WHERE Clause

The information of the WHERE clause is parsed and stored in a tree structure called a parse tree. For example, the parse tree structure representing the condition

WHERE sex = ‘M’ and salary > 50000

is shown in Figure 3.3.

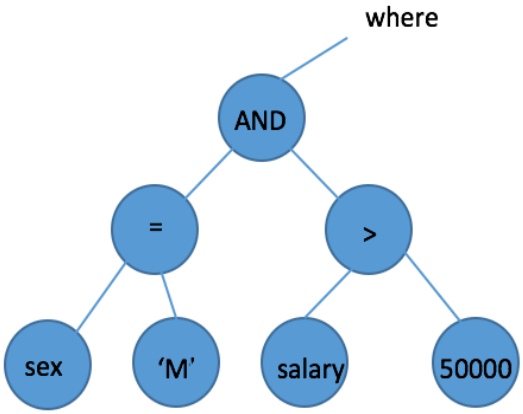


Figure 3.3: Parsing Tree 1

Each node of the parse tree contains 2 fields:

(1) *type*: contains an integer code that indicates the type of the node

(e.g. 136 = AND, 172 = OR, 128 = ATTRIBUTE, 131 = STRING CONSTANT …., etc.)

(2) *value*: contain the values used to represent information stored in the node

Depending of the value of the type variable, we must store different information in the *value* variable. For instance, if *type* = INTEGER (constant), the variable *value* will contain an integer. But if *type* = ATTRIBUTE, the variable *value* will contain a relation name and an attribute name. The appropriate data type for the *value* variable is therefore a C union type. The following is the definition of a node used in the project:

```
struct MyNode {
   int type;    /* Can be one of PLUS, MINUS, MULT, DIV, Negate  */
                /* Attr, INT, FLOAT, STring, AND, OR, NOT, IN... */
                /* RelOp: >, >=, .... */
   union {
    char   val[50];     // INT, FLOAT, STRING, Boolean (T,F) constants
    struct {
      char   relaName[50];     // Attribute
      char   attrName[50];
    } attr;
    struct MyNode *p[2];     // Operations
                             // WHen type = PLUS, MINUS, MULT, AND. OR...
                             // NOT: use only p[0]
                             // Negate: use only p[0]
   struct {
     struct Select_Type *obj;
     struct MyNode *p[1];
   } sel;        //link to a select object
   } value;
};
```

Figure 3.4: Structure of Node

We used the *recursive descend technique* to implement the parsing of the grammar in Table 3.6. The top level function is called BE( ) and parses a <b-expression>:

<b-expression> ::= <b-term> [OR<b-term>]*

A <b-expression> consists of a <b-term> (boolean term) followed by zero or more "OR <b-term>" expressions. The function BE( ) returns the pointer to the root node of the parse tree. This pointer is stored in a program variable called WHERE. If the query does not contain a WHERE clause, the WHERE variable is set to NULL –which represents a "true" condition. Because "<b-expression> ::= <b-term> [OR <b-term>]*", the BE( ) function will first call the BT( ) function that parses a <b-term> (Boolean term). The function BT( ) will also return a pointer to its parse tree. BE( ) will save the return result of BT() in a help variable *h1*. Because in a <b-expression> <b-term> can be followed by zero or more "OR<b-term>" expressions, the BE( ) function will check if the next token in the input is equal to OR. If the next token is not equal to OR, BE( ) will return the value in *h1* and the <b-expression> has been parsed completely. Otherwise (i.e.,

the next token is OR), the function BE( ) will call BT( ) again to obtain the second <b-term> (let us call this b-term2) and constructs the following tree:

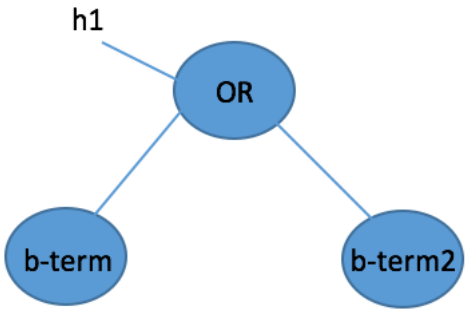


Figure 3.5: Parsing B-expression 1

The root of this tree is stored in the help variable *h1*. The BE( ) function will then repeat and check if the next token in the input is equal to OR. If the token is not equal to OR, the BE( ) function will return the parse tree in Figure 3.5. Otherwise, BE( ) will call BT() again (b-term3) and constructs the following parse tree:

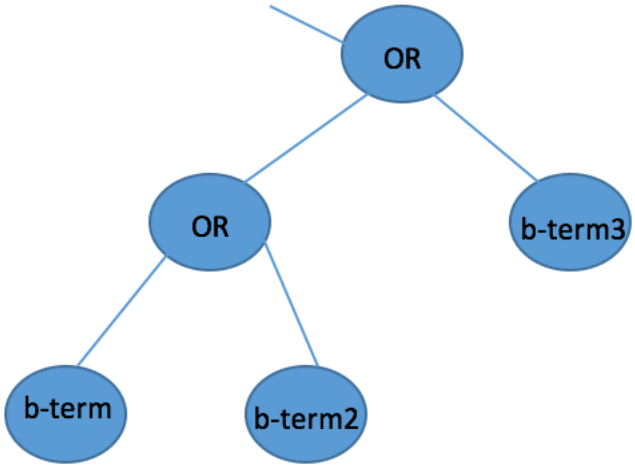


Figure 3.6: Parsing B-expression 2

And so on.

The function BT( ) parses a <b-term> (boolean term) and has the same structure as BE( ), except that we check for the token AND to decide if we need to repeat because the grammar for <b-term> is:

<b-term> ::= <not-factor> [AND <not-factor>]*

The <not-factor> element is handled by the NBF( ) function in the parser. The grammar rule "<not-factor> ::= [NOT]<b-factor>" specifies the optional keyword NOT. This rule is processed by reading a token and check against NOT. If the token is equal to NOT, we will call BF( ) to parse a <b-factor> (Boolean factor). However, if the next token is not equal to NOT, then we must return the token back into the input stream before we can call BF() (to parse the <b-factor>). This is accomplished by using the *yyless*(0) function.

The next grammar rule that we must process is a Boolean factor which is processed by the BF( ) function:

<b-factor> ::= <b-variable> | <expression> <relop> <expression>
| attribute IN (query)

Since a <b-variable> and an attribute are identifiers, they can be considered as instances of <expression>. In the parser, we implement the following modified rule for <b-factor> to simplify the implementation:

<b-factor> ::= <expression> <relop> <expression> | attribute IN (query)

According to the modified rule, the BF( ) function will first parse an <expression> by calling the function E( ). We save the result from E( ) in a help variable *h1*. Then we check if the next token is a <relop> (relational operator: <, <=, >, >=, = , !=) or the keyword **IN**. The processing proceeds as follows:

(1) If next token is a <relop>, we call E( ) again and obtain the second <expression> and return the following parse tree:

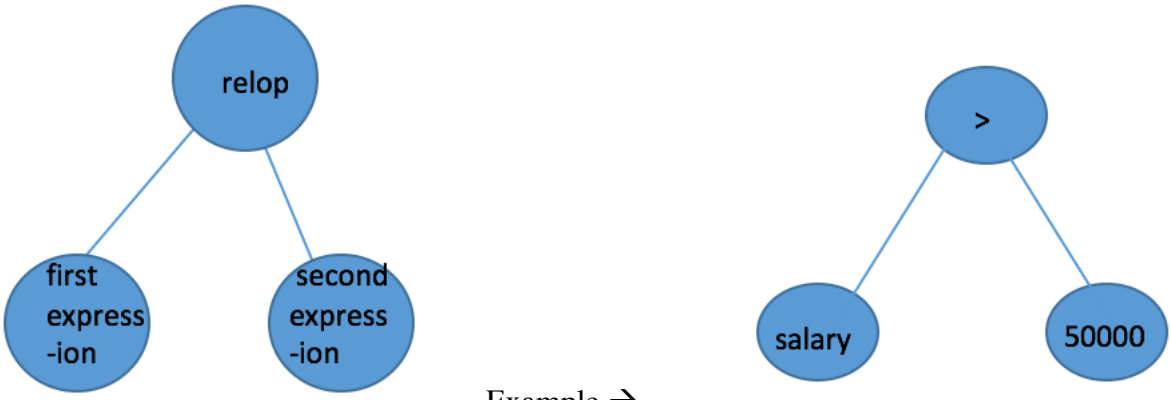


Figure 3.7: Parsing B-factor 1

(2) If next token is the keyword IN, we recursively invoke the SQL parser function (**SelectObject()**) an return the following parse tree:

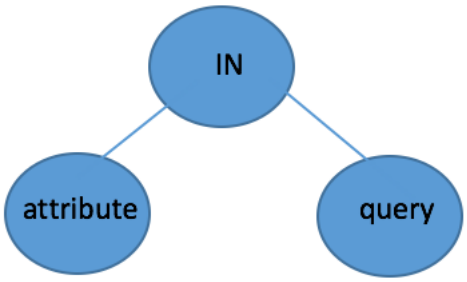


Figure 3.8: Parsing B-factor 2

(3) Otherwise, we return the <expression> as a Boolean expression. This modified grammar rule allows us to use the recursive descent method to process a Boolean factor. (In fact, this rule is used in the C- programming language because an expression in C is also a Boolean expression).

The remaining grammar rules define the syntax of arithmetic expressions. The function E( ) in the parser processes the grammar rule:

<expression> ::= <term> [<ADD><term>]*

<ADD> represents the addition operation + or -.

The function E( ) returns a tree representing arithmetic operations involving additions and subtractions and its structure is similar to the BE( ) function. BE( ) first call the function T( ) that parses a <term>. The function T( ) will return a pointer to its parse tree and E( ) will save the return value in a help variable *h1*.

Because a <term> can be followed by one or more "<ADD><term>" expressions:

<expr> ::= <term>[<ADD><term>]*

E( ) will check if the next token in the input is an <ADD> operator (+ or -). If the next token is not equal to <ADD>, E( ) will return the value in variable *h1* and its execution is completed. Otherwise (i.e, the next token is an <ADD> operation), the function E( ) will call T( ) again and obtain the second <term> (term2) and construct the following parse tree:

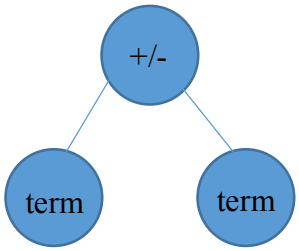


Figure 3.9: Parsing Expression

Since "[<ADD><term>]" can repeat just like a <b-expression>, we handle the repetition in the same way as in the BE( ) function.

The function T( ) processes the grammar rule "<term> ::= <factor>[<MULTIPLY><factor>]*" and has the same structure as E( ). The differences are: (1) T( ) looks for multiplication (*) or a division operation (/) and (2) T( ) will call F( ) to process a <factor>.

The function F( ) recognizes the grammar rule:

<factor> ::= <number> | <string constant> | <identifier> | (<b-expression>)
| <ADD> <factor>

F( ) handles the most basic elements of the parse tree. The first case is a number which is stored as the value in a node. The second case is a string constant (e.g. 'abc'). The third case is an identifier that represents an attribute (e.g. salary, SSN, etc). When the next token is '(', the <factor> is a bracketed expression (e.g.:(a+b)) and we call the function E( ) to parse the expression. Finally, if the next token is an <ADD> (i.e. '+' or '-'), we create a unary operator node for the <ADD> and repeat to parse <factor> as its operand. For example, the expression "-a+-b" is represented as:

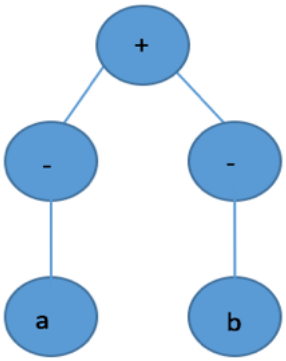


Figure 3.10: Parsing Expression Example

3.7 **Converting to Relational Algebra**

After parsing a SQL query, we use the information in the parse tree to construct a corresponding Relational Algebra expression tree. The Relation Algebra query has the following structure:

$$\boldsymbol{\pi_{a_1,a_2,a_3}}(\boldsymbol{\sigma_C}(\ R_1\ \text{X}\ R_2\ \text{X}\ R_3\ ))$$

and can be viewed as a tree structure [2]:

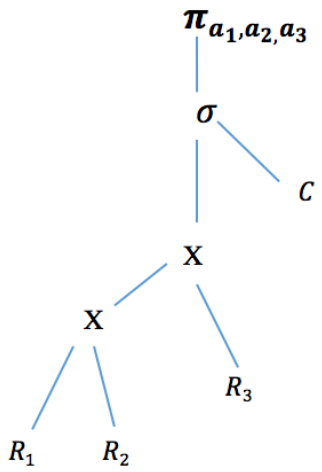


Figure 3.11: Relational Algebra Tree

```
struct RAN {
  int type;
  struct {
  struct RAN *left;
  struct RAN *right;

  //store
  struct MyNode *where_r;
  struct AttrType attrlist[10]; ////////////////////////
  int attrnum;
  //struct RelaType *relalist;
  struct {
      char   relaName[50];     // Attribute
      char   alias[50];
    } relation;
  }value;//end of value
};
```

Figure 3.12: Structure of Relational Algebra Node

We use a different node structure to represent a Relational Algebra tree. The structure of a Relational Algebra node is given in Figure 3.11. The variable *type* indicates the type of a node, which represents the Relational Algebra operation (e.g.: Projection $\boldsymbol{\pi}$, selection $\boldsymbol{\sigma}$ or Cartesian Product **X**). Depending on the value of the *type* variable, different variables in the variable *value* will represent the operands for the Relational Algebra operation. The projection operation($\boldsymbol{\pi}$) uses the *left* variable to store the input operand and the *attrlist* to store the list of projection attributes. The selection operation ($\boldsymbol{\sigma}$) also uses the *left* variable to store its input relation but uses the *where* variable to store the selection condition (a Boolean expression). This Boolean expression is the same as the one returned by the SQL parser for the WHERE clause. The Cartesian Product (**X**) has two input relations which are stored in the variables *left* and *right*.

Chapter 4

Testing and Evaluation

4.1 Sample Outputs

In this Chapter, we present the results of a number of test inputs (queries) and their resulting SQL parse tress and the corresponding Relational Algebra expression trees. The first example is a SQL query without any Boolean condition:

Example (1):

SELECT SSN, salary
FROM employee

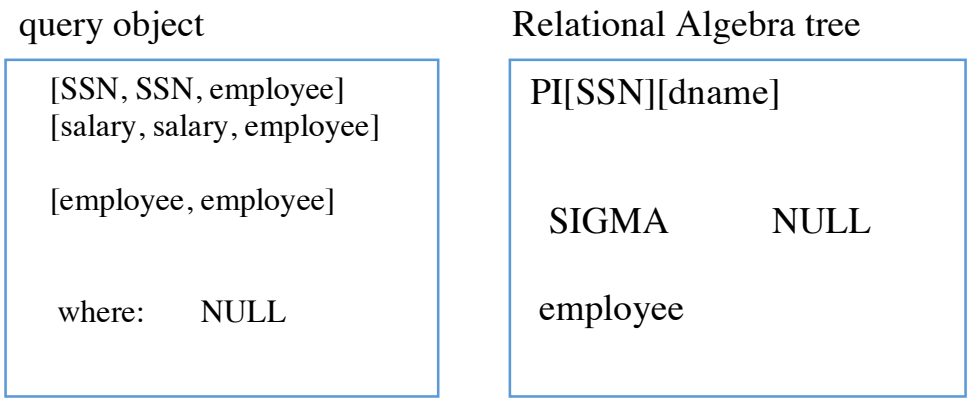


Figure 4.1: Output 1

The query object contains the parse information. The content of the variables in the query object is as follows:

attribute list (array)

attr 1	Name	Alias	Relation
attr 2			

relation list (array)

rel 1	Name	Alias
rel 2		

WHERE = NULL (root of the boolean expression)

The WHERE tree in this example is empty. The Relational Algebra tree in Figure 4.1 consists of the node $\boldsymbol{\pi}(\boldsymbol{PI})$, $\boldsymbol{\sigma}$ $(\boldsymbol{SIGMA})$, and **X** and is printed from left to right (i.e.: rotated view). The Relational Algebra tree shown in the figure is the following tree:

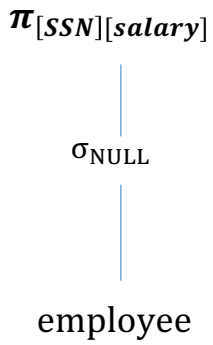


Figure 4.2: Relational Algebra Tree

The second example shows the parse result of a SELECT query with a simple Boolean condition.

Example (2):

SELECT SSN ID, dname N
FROM department, employee
where dno = dnumber

query object

[ssn, ID, employee]
[dname, N, department]

[department, department]
[employee, employee]

where: = employee.dno / department.dnumber

Relational Algebra tree

PI[SSN][dname]

SIGMA = employee. dno / department. dnumber

X: department / employee

Figure 4.3 Output 2

The WHERE expression in the SQL parse-tree is also output from left to right and represent the following boolean expression:

=

employee. dno department. dnumber

In example 3, we show the parse result of a compound condition.

Example (3):

```
SELECT SSN ID, dname dpt
FROM department,employee
WHER dno = dnumber
        AND employee.sex = 'female'
        AND department !='Research'
```

query object

[SSN, ID, employee]
[dname, dpt, department]

[department, department]
[employee, employee]

employee.dno
=
department.dnumber
AND
employee. Sex
=
‘female’
where: AND
department.dname
!=
‘Research’

Relational Algebra tree

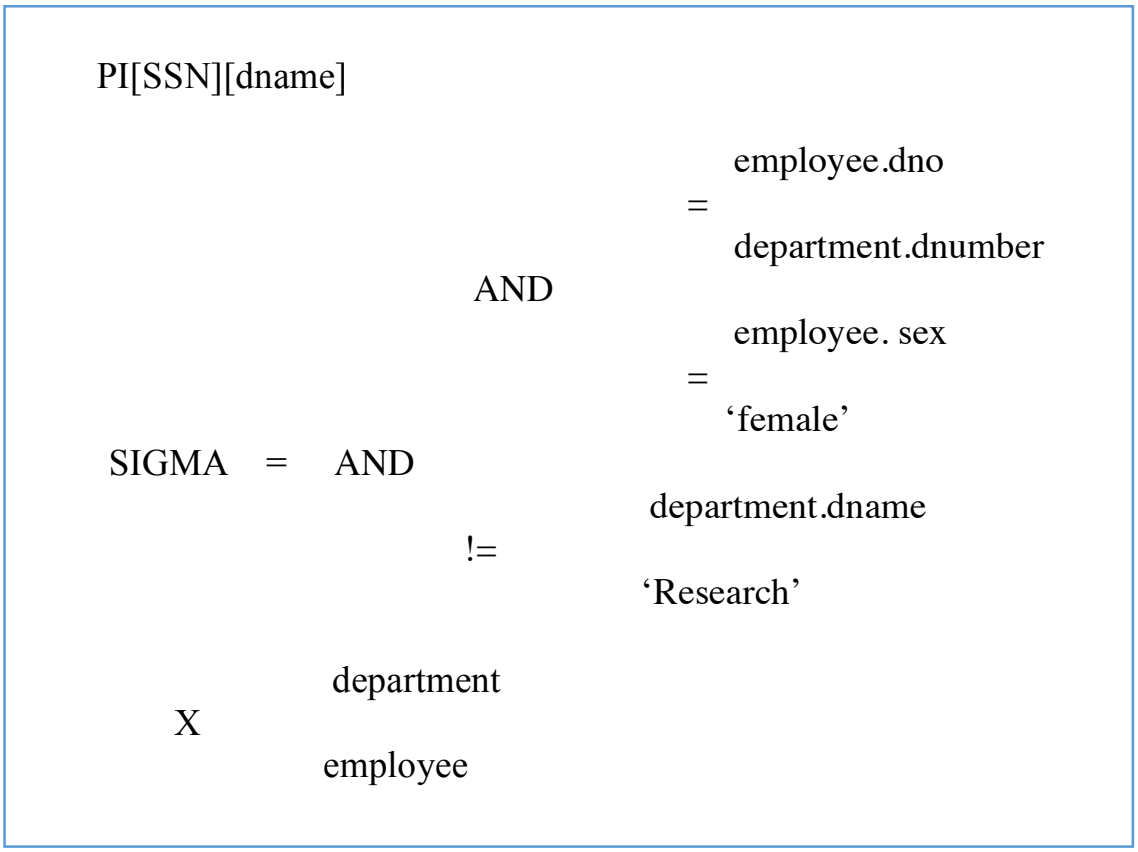


Figure 4.4: Output 3

Finally, example 4 shows the parse tree of a nested query output by the program.

Example (4):

```
SELECT SSN, dname
FROM department, employee
WHER dname = dno
        AND SSN IN (SELECT SSN
                    FROM employee
                     WHERE SSN > 729740169)
```

query object

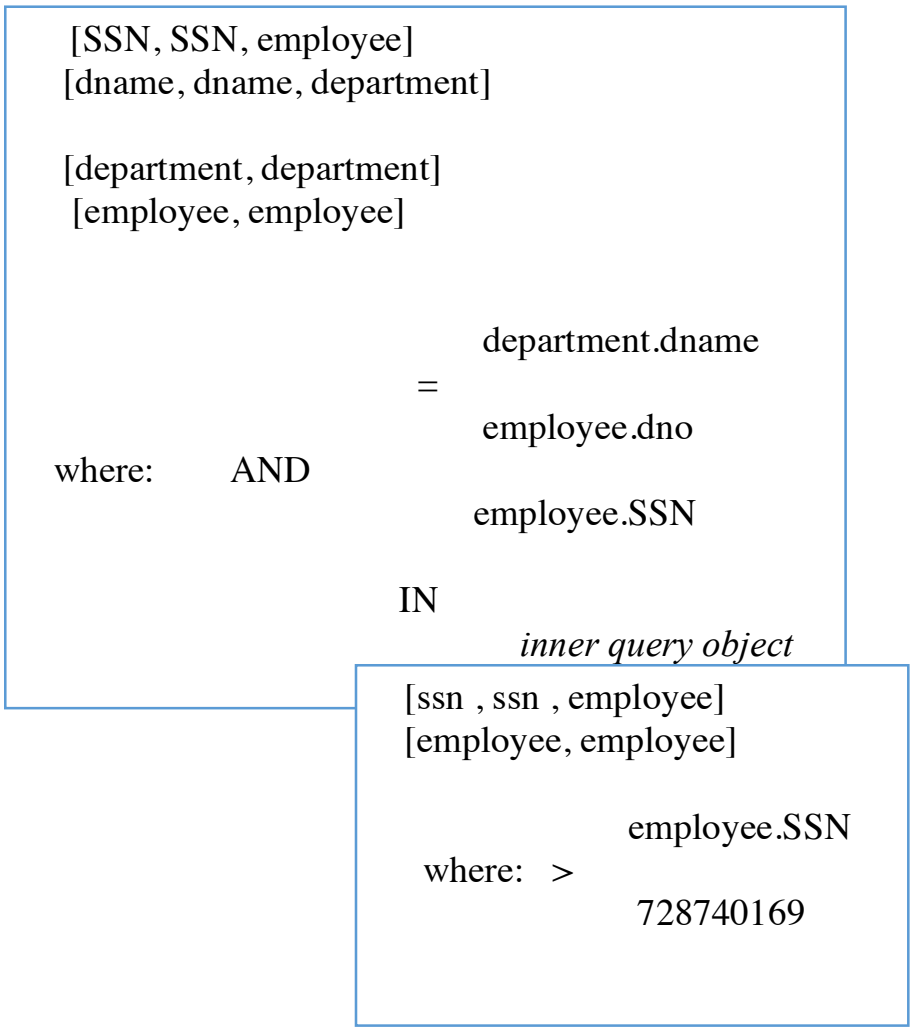


Figure 4.5: Output 4

The Relational Algebra tree for this example is not constructed because this part was not implemented due to time constraints. We will discuss its implementation as part of the improvements in Chapter 5.

4.2 **Limitations**

I have tested the program of translating SQL queries using the instances that represent the variation of the SQL query formats. The number of the instances that I input to test the program was 25. I have used three relational databases. The maximum number of attributes in the relation model is ten.

The program handles the following errors: 1) The attribute and the relation are not found in the relations. After getting the tokens from the SELECT command, the program then checks the names of the relation and the attribute against the metadata. If the name is not found, then the program will identify as an error and ask for the input of another query. Otherwise, it will continue to parse the WHERE clause. 2) The Where clause is not construct logically. For example, the AND operator requires two operands. The program will return NULL and no Boolean expression is parsed.

The project only covers part of the expressive power of SQL. It handles SQL queries in the format of "SELECT <attribute_list> FROM <relation_list> [WHERE <boolean_condition>]" or "SELECT <attribute_list> FROM <relation_list>[WHERE attribute IN (SQL-query)]". The WHERE clause can also be in the formats mentioned in section 3.6. In addition, the SQL query can add the clauses "GROUP BY <gb_attr> [HAVING <hav_condition>]". GROUP BY clause groups a set of rows by values of columns and HAVING clause restricts the groups to satisfy certain conditions.

Chapter 5

Project Evaluation and Future Work

5.1 Project Goal and Assessment

The goal of the thesis project was to gain deeper understanding in Database Systems. As part of the honors requirement, I am taking the graduate Advanced Database Systems (CS 554) course. However, the course does not discuss the query translation process in detail. The thesis project aims to supplement this knowledge.

In this project, I have learned how "sentences" are represented by grammar rules and built a parser to recognize a subset of the grammar rules of SQL. In the project, I became more aware of the fact that solving problems effectively requires the choice of the appropriate data structures to store the necessary information. For instance, I learned that a parse tree is the most effective way to represent expressions.

Without having taken a prior course in Compilers, I have now discovered the importance of parsing techniques while learning to develop the SQL parser to recognize a simplified context-free SQL grammar subset. I have learned the *recursive descent parsing technique* that uses the idea of *a look-ahead token* to decide what to do next. I have also learned how to take into account the possible derivation rules.

I have improved my ability to take into account all the possible situations when trying to solve a problem. For instance, when I designed a structure to represent a node in the parse tree, I

learned to distinguish the situations when it is an identifier, a number, a string constant, and an operation and how to store their corresponding value/operands.

5.2 Improvement

Due to time constraints, only a subset of the SQL grammar was implemented. The formats of WHERE that were omitted are (mentioned in section 3.6):

(1) attribute rel-op ALL (SQL-query)

(2) attribute rel-op ANY (SQL-query)

(3) EXISTS (SQL -query)

(4) IS NULL

(5) attribute LIKE “pattern”

All these clauses are Boolean factors and the code to process these clauses must be added in the BF( ) function.

For clauses (1) and (2), the parsing algorithm is similar to the format “attribute IN (SQL-query)”. To parse these formats, we should detect the token representing a relation operator followed by the token ALL or ANY. Then, we parse the sub-query just like how we handle the “IN(SQL-query)” clause. Clause (3) requires identifying the token EXISTS and parsing a sub-query. Clause (4) and (5) are relatively easy: we only need to detect the tokens of IS and NULL or an identifier (attribute name), the token LIKE and a string constant (pattern).

Another improvement that we can make to the project is in the processing of the sub-query parsing. Currently, we can only handle *uncorrelated* sub-queries, i.e.: attribute names used in the

inner query must belong to relations in the inner query. In general, attribute in the sub-query can belong to relations in the inner *or* the outer query. For example:

SELECT name

FROM employee E

WHERE salary >= ALL (SELECT salary

FROM employee

WHERE dno = E.dno)

The attribute **E.dno** in the inner query belongs to the relation **employee E** in the outer query. Furthermore, we have limited the query nesting to two levels. In general, SQL query can have multiple nesting levels. Processing such a multi-level nested query will require each inner query object to refer to its parent query object. For example, the query is stored as:

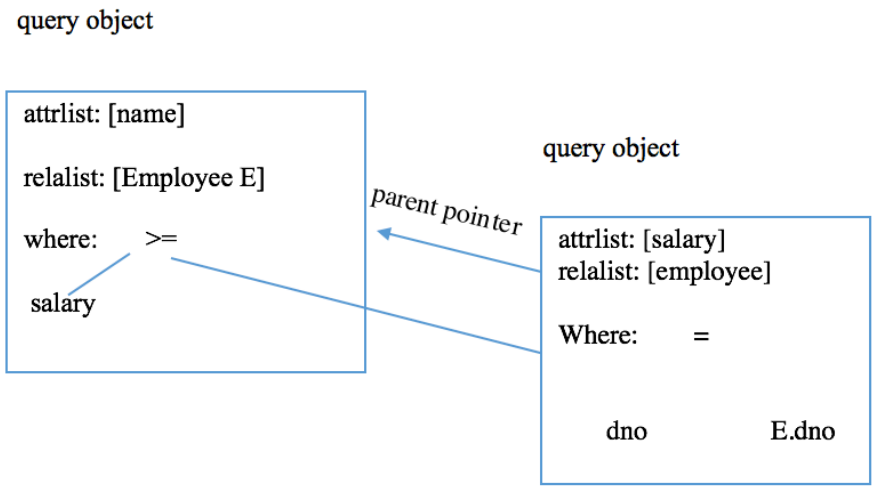


Figure 5.1: Query Object that Contains a Correlated Sub-query

To identify the relation for an attribute, we must traverse the query from inside out using the parent pointers. For example, the attribute **E.dno** is not found in the inner query, so we will use the parent pointer to locate the parent query object. When we search in the outer query object, we will find the relation **employee E** where the attribute **E.dno** belongs.

In the project, we have implemented parsing of sub-query constructs in the form "attribute IN (SQL-query)". Due to time constraints, the translation to the Relational Algebra tree was not implemented. The translation process is discussed in the CS554 (Advanced Database System) course. We will first build a Relational Algebra tree in Figure 5.2 which uses a <R-cond> expression.

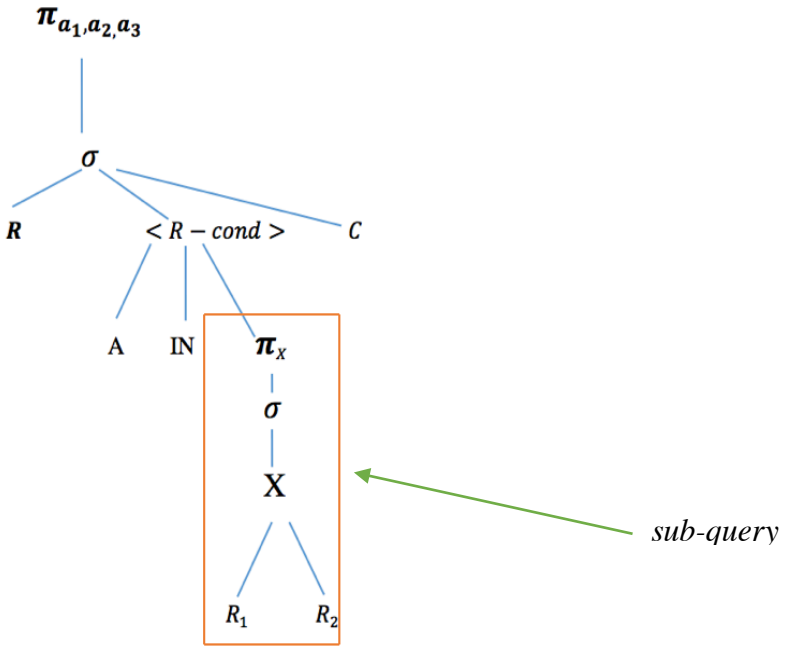


Figure 5.2: Relational Algebra Tree Manipulation 1

The <R-cond> is an auxiliary form to help us convert an SQL-parse tree into a Relational Algebra tree. The tree in Figure 5.2 is then transformed (rewritten) into a valid Relational Algebra tree using a tree transformation operation. Figure 5.4 illustrates the transformation to obtain a valid Relational Algebra tree.

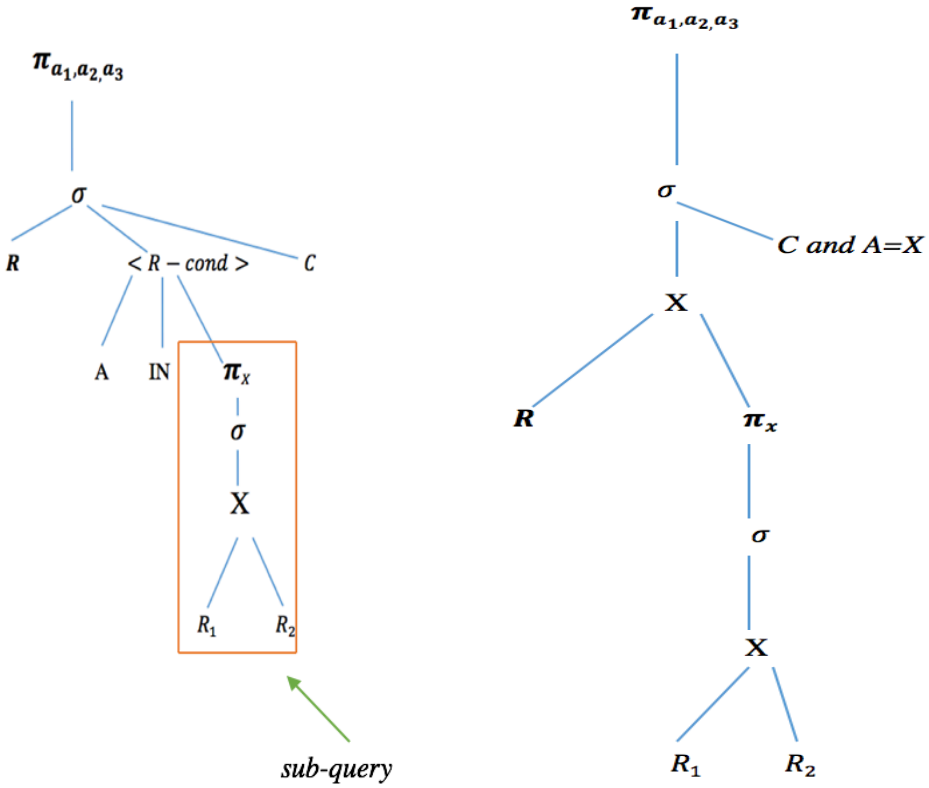


Figure 5.3: Relational Algebra Tree Manipulation 2

Considering the following simple SQL nested query:

SELECT name

FROM employee

WHERE SSN IN (SELECT ESSN

FROM dependent)

The SQL-parser of the project will construct the SQL parse tree in Figure 5.4

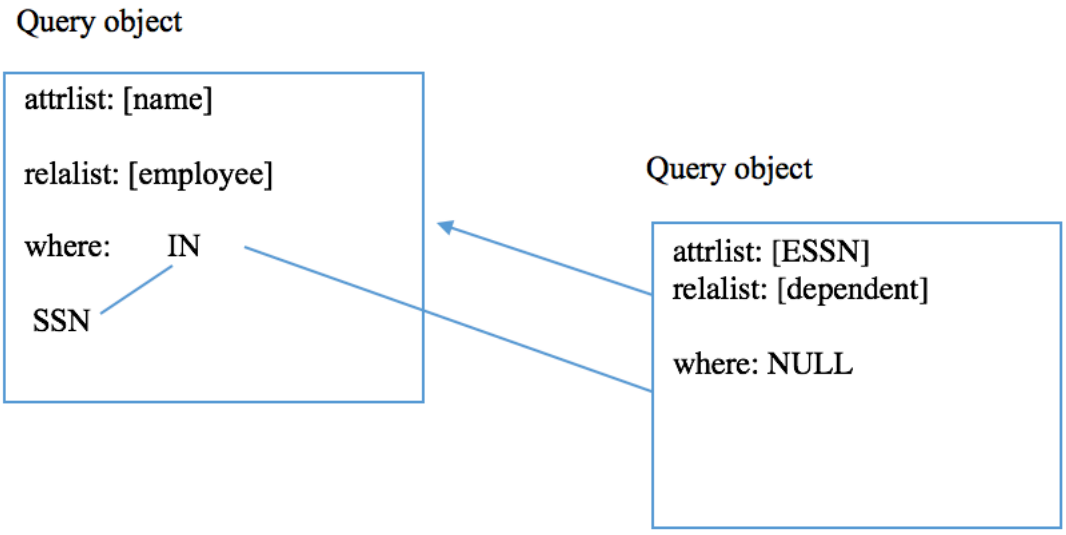


Figure 5.4: Query Object that Contains *attribute IN (sub-query)*

We first construct the auxiliary Relational Algebra tree with an <R-cond> in Figure 5.5.

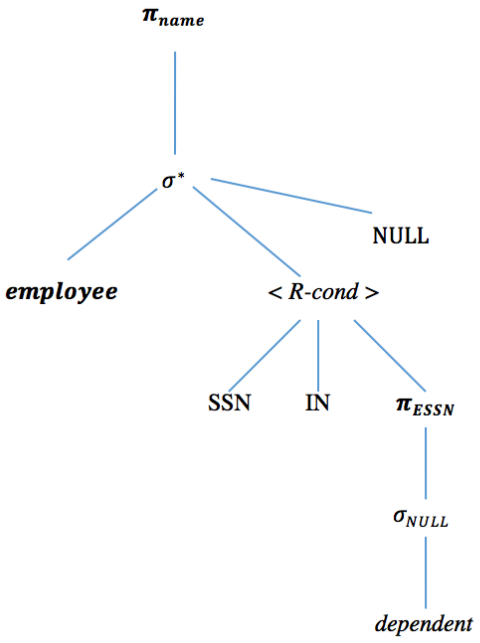


Figure 5.5: Relational Algebra Auxiliary Tree Example

This auxiliary tree can be constructed from the SQL-parse tree in a straight forward manner.

After applying the tree transformation, we would obtain the Relational Algebra tree in Figure 5.6.

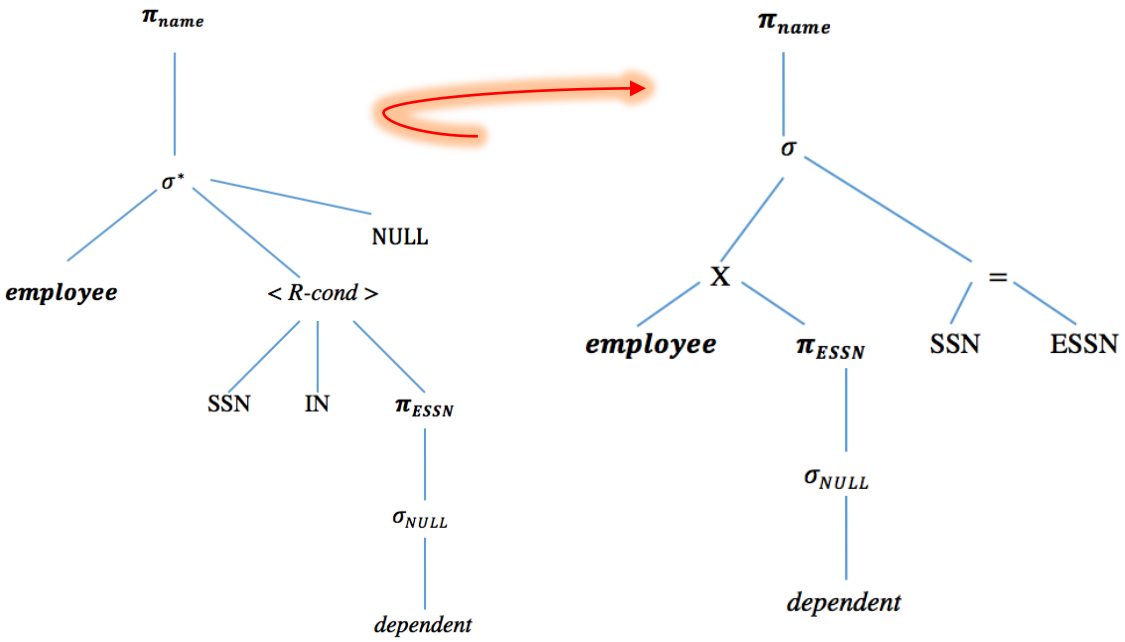


Figure 5.6: Relational Algebra Tree Manipulation Example

Bibliography

[1] Elmasri, Ramez, and Shamkant B. Navathe. "Chapter 3 The Relational Data Model and Relational Database Constraints." Fundamentals of Database Systems. Boston: Pearson, 2010. N. pag. Print.

[2] Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. Database Systems: The Complete Book. Harlow, Essex: Pearson, 2008. Print.

[3] Appel, Andrew W., and Maia Ginsburg. Modern Compiler Implementation in C. Cambridge: Cambridge UP, 1998. Print.