TYPE INFERENCE IN SQL

WEISHENG LIN

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

APRIL 2004

Your file Votre référence
ISBN: 0-612-91069-5
Our file Notre référence
ISBN: 0-612-91069-5

Abstract

Type Inference in SQL

Weisheng Lin

Type inference is an important concept in programming languages. In this Thesis, we study this problem and propose a framework for type inference in SQL, the database programming language for relational databases such as Oracle and Sybase. We consider a context-free grammar G_{SQL} which covers the core features of the standard SQL. We add *semantic rules* to G_{SQL}, following Knuth's method of "attribute grammars", to capture the set of schemas for which a query $q \in L(G_{SQL})$ is well-defined. We show that G_{SQL} is unambiguous and that our attribute grammar is non-circular. The set of schemas of a query is usually infinite. To finitely represent this set, we introduce *schema tableaux*, a variation of a well-known tool from database theory. By defining another attribute grammar for G_{SQL}, we show that the set of schemas of a query $q \in L(G_{SQL})$ can be finitely represented as a tableau which can be effectively computed given q as input. We discuss applications of our type inference methodology, and as a case study, we apply it on the suite of TPC Benchmark^{TM} H queries, which has industry-wide relevance and a high degree of complexity. The experiments indicate the methodology in useful in practice, particularly in the context of database schema comprehension.

Acknowledgments

I express deep gratitude to my supervisors Drs. G. Grahne and N. Shiri for their guidance and support throughout my research at Concordia University.

I am thankful to my colleagues in the database lab for sharing time and thought. They are Zhi Hong Zheng, Xiao Long, Yu Song, Xi Deng, Tong Zhang, Hong Bing Zhang, Bei Hua Zhang and Ali Kiani. I am especially grateful to Ali Kiani for many helpful conversations.

Last but not the least, I would like to thank my girl friend, Xuefei, for having been so patient and supportive to me.

Contents

List of Figures **vii**

List of Tables **viii**

1 Introduction **1**

1.1 Problem Statement 3

1.2 Motivation 7

1.3 Contributions 9

1.4 Outline of the Thesis 11

2 Background Review **13**

2.1 Related Work 13

2.2 Type Inference in the Polymorphic Relational Algebra 15

2.3 Solving Systems of Set Equations 19

2.4 From Relational Algebra to SQL 21

3 Proposal **23**

3.1 Schema 24

3.2 Schema Tableaux 36

3.3 Rules to Compute Schema Tableaux 43

3.4 Implementation of Operator Merge 46

3.5 Extension of Grammar G_{SQL} 52

4 Applications and Experiments 53

4.1 Application: Discovering the Database 54

4.2 Application: Stored Procedure Maintenance 60

4.3 Experiments on TPC-H 64

5 Conclusions and Future Work 81

Bibliography 83

List of Figures

1 Typing Rules of the Relational Algebra 17

2 Directed Graphs of the Rules 32

3 Non Circularity of Semantic Rules of G_{SQL} 33

4 From Queries to Tableaux 55

5 An Illustrative Example 57

6 Schema Tableau of Query 1 58

7 Schema Tableau of Query 2 59

8 Schema Tableau of Query 3 59

9 Schema tableau of Queries 1, 2, and 3 59

10 Database Schema Comprehension – Example 60

11 Maintenance of Stored Procedures 63

12 TPC-H Schema 67

13 Schema Tableau of Query 1 69

14 Schema Tableau of Query 2 71

15 Schema Tableau of Query 3 73

16 TPC-H Database Schema Comprehension 77

List of Tables

1 Summary of All Relations 74

2 Relationships Discovery 75

3 Summary of Relation Part 78

4 Summary of Relation Supplier 78

5 Summary of Relation Partsupp 78

6 Summary of Relation Lineitem 79

7 Summary of Relation Costumer 79

8 Summary of Relation Orders 80

9 Summary of Relation Nation 80

10 Summary of Relation Region 80

Chapter 1

Introduction

With the development of programming languages, type theory has become increasingly important. In the context of programming languages, a type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute [12].

A data type in programming language is a set of data with values having certain properties that are predefined. Examples of data types are: integer, character, string. Programmers need to declare the data type of every data object for most programming languages, and most database systems require the user to specify the type of each data field.

One of the major developments in programming is polymorphic type inference. Type polymorphism allows code to handle data objects regardless of their data types. Type inference automatically assigns the output and input types to a function if it is not given[20]. Type inference is an extensively studied topic and used in industrial-strength functional programming languages as well such as SML/NJ [4].

The form of declaration of a relation schema consists of the keyword `Create Table` followed by the name of the relation and a parenthesised list of the attribute names and their types. One example is as follows:

```
Create Table R(
   A  CHAR(2),
   B Char(20),
   C Date,
   D int
  );
```

The operators of SQL are polymorphic. We can take the cross product of any two tables, regardless of their schema. We can perform a comparison between two attributes, as long as they are comparable, namely they are either of the same type, or in a same type hierarchy. Types *int* and *double* are different, but they can be regarded as subtypes of some more general type, for example, *number* and hence comparable. In SQL expression `A = B`, the attributes A and B can be both of *char*, or one is *int* and the other of *double*, for instance. When the SQL expression becomes complex, all these type conditions become more involved. Consider the following SQL expression:

```
Select A, B
From R, S
Where (B, C) in (
        Select (D,E)
        From S
        )
```

For this expression to be well-typed, attribute A must be in relation R or S but not in both, and so is attribute B; attributes D and E must be both in relation s. Moreover, B must be comparable with D, and C must be comparable with E.

1.1 Problem Statement

A natural question thus arises: given an SQL expression e, under which database schema is e well-typed? And what is the result relation schema of e under each of these schemas? Obviously, this is the SQL version of the classical type inference problem.

Doing type inference for a language involves setting up two things. Firstly, we need a system of type rules that allows us to derive the output type of a program given types for its input parameters; under these assignments the program is said to be well-typed. Second, we need a formalism of type assignments, as well as an output type for each type assignment in the family. Every typable program should have a principal type formula, which defines all type assignments under which the program is well-typed, as well as the output type of the program under each of these assignments. The task then is to come up with a type inference algorithm that will compute the principal type for any given program [4].

The issue that given a relational algebra expression e, under which database schemas e is well-typed was investigated recently [4]. In relational algebra, an attribute name says all about the attribute. Hence multiple occurrences of an attribute in different relations of a database refer to the same concept including its type. But this may or may not be the case for SQL. For the expression $R \cup \pi_{A,B}(S)$ to be valid, for example, the schema of R must be $R(A, B)$. In SQL, however, due to the explicit introduction of data types, attribute name is associated with its data type.

Let us examine the SQL counterpart of the previous example,

```
Select * From R Union Select A, B From S.
```

For this SQL statement to be valid, attributes A and B are no longer required to appear in relation R. It suffices for the validity of this SQL statement that R is a binary relation having the first attribute of type of $S.A$, and the second of type of $S.B$. Moreover, even if binary relation R has attributes A and B, the SQL statement could be illegal. It is the case when $R.A$ and $S.A$ are of different types, and/or when so are $R.B$ and $S.B$, in spite of the fact that it is unlikely to happen in a well-designed database application.

There is another issue, too. In relational algebra, the collection of attributes of a relation is regarded as a set. Conceptually, it is still true for SQL. Practically, however, attributes are listed with some order that should be followed by some operators. Hence in relational algebra, we can take the union of relations $R(A, B)$ and $S(B, A)$. The polymorphic operator $\cup$ produces as output relation ($R.A \cup S.A$, $R.B \cup S.B$). Now let us see what happen if we perform the similar operation in SQL.

First we create two tables R and S by

```
Create Table R (A int, B char(1)),
```

and

```
Create Tables (B char(1), A int).
```

If we try to run the SQL query

```
Select * From R Union Select * From S.
```

This is regarded as an illegal operation, although both tables R and S have attributes A with the same type, and B with the same type, too. This is because when the tables are created, each attribute is assigned to a column according to the order as they appear in the `Create Table` expression.

The following is what we get running this test on Oracle.

```
SQL> Create Table R(A int, B char(1));
Table created.
SQL> Create Table S(B char(1), A int);
Table created.
SQL> Select * From R Union Select * From S;
Select * From R Union Select * From S
       *
ERROR at line 1:
ORA-01790: expression must have same data type as corresponding
expression
```

The type of an output attribute inherits the type of the attribute where it is projected. In this SQL example, the output is A and B. And the type of A is the same as A in the table R or S. And so is B. However, the case of union is an exception.

In the following example, we show that when two attributes with comparable types are arguments for union, the output type is a type, of which these two attributes are subtypes.

```
SQL> Create Table R(A varchar(10), B number(20,0));
Table created.
SQL> Create Table S(C char(1), D number(18,3));
Table created.
SQL> Create View RS As
  2  (Select * From R Union Select * From S);
View created.
SQL> Create View SR As
  2  (Select * From S Union Select * From R);
View created.
SQL> desc R;
 Name                     Null?    Type
 ------------------------ -------- ---------------
 A                                 VARCHAR2(10)
 B                                 NUMBER(20)
SQL> desc S;
 Name                     Null?    Type
 ------------------------ -------- ---------------
 C                                 CHAR(1)
 D                                 NUMBER(18,3)
SQL> desc RS;
 Name                     Null?    Type
 ------------------------ -------- ---------------
 A                                 VARCHAR2(10)
 B                                              NUMBER

SQL> desc SR;

 Name                           Null?    Type
 ------------------------------ -------- ---------------
 C                                       VARCHAR2(10)
 D                                       NUMBER
```

Related work proposes interesting treatment on type inference in relational algebra, yet applying it in SQL remains to be addressed. Furthermore, SQL specific features such as aggregation provide extra knowledge of attributes and the conditions on them, but are not captured by the primitive form of relational algebra, either.

Since these SQL specific features are missing in relational algebra, there is a need to investigate the type inference issue for SQL. We believe it is a natural approach to collect and infer attribute constraints and type information from SQL statements directly, as the rest of this thesis will demonstrate.

1.2 Motivation

One main motivation for this work is fundamental and theoretical. SQL is a specialised programming language, consequently important ideas in programming languages such as type inference should be applied and adapted to SQL. There are also other scenarios in database and engineering that add weight to the motivation.

Imagine a crucial application which relies on some antiquated RDBMS. For some reasons, say, the vendor ran out of business, and the system is no more supported.

Even worse, it crashed before the user has the chance to migrate to a modern DBMS. Since it is critical, the user cannot afford to lose the data and have to recover the system as much as possible. Fortunately, there is a reasonably comprehensive set of queries available, for example, embedded in some host language. In such a situation, our work could come to rescue. Incorporated with other techniques of reverse engineering, we believe the system can be reconstructed to some extent.

Not only useful in antiquated systems, our work is also instrumental in recent trends of database programming. Many business applications centre increasingly around a database. The key to such applications is ensuring that all database interaction are both efficient and secure. Stored procedures help developers accomplish both of these goals and also accomplish greater abstraction between the application code and database details [10]. Stored procedure is a set of SQL statements that is assigned a name and stored in a database in compiled form so that it can be shared by a number of programs.

Stored procedures can contain not only simple select, insert, update, and delete statements, but also can encapsulate procedural code. Encapsulating the database interaction provides a number of benefits, including

1. It simplifies the coding. Instead of using a large query, one just call a stored procedure.

2. It allows the user to optimise the queries in a central location.

3. It achieves greater security.

Due to its importance, stored procedure is one of the basic building blocks in database programming. However, a large number of stored procedures in database pose new challenge as well. When a user calls a stored procedure, the server does not always create a new data access plan when retrieving information. Clearly, when a procedure references a database table and the structure of the table changes, the plan has to change, too. Unfortunately, the server may not keep a complete database of dependencies and is not able to determine which procedures must be recompiled when a structural change takes place. If one database programmer finds herself in a situation in which she has a considerable number of stored procedures and her table structures change quite often, say she is prototyping a new system and is using a trial-and-error approach, keeping track of all the dependencies can be pretty painful [19]. Therefore, a tool that examines the validity of stored procedure when the schema changes is very helpful. Our work fits in nicely this scenario.

1.3 Contributions

The contributions of our work are as follows:

1. Type inference is an important concept in programming languages. SQL is a specialised language, but to the best of our knowledge, theory of type inference in SQL is missing in the literature. In this research, we make a first attempt and investigate this problem.

2. We propose a framework in which database schema is formally defined, introduce a grammar G_{SQL} that covers the core features of the standard SQL, and assign attributes to syntactic rules by a set of semantics rules.

3. We develop a formalism, schema tableaux, which is specially tuned for specifying incomplete information of database schemas. We also define an important operator merge on schema tableaux.

4. We establish that schemas satisfying nodes of derivation trees can be represented by schema tableaux. Furthermore, we devise rules to compute schema tableaux associated with nodes of the derivation trees of queries generated by G_{SQL}.

5. We apply the proposed framework on the suite of queries of TPC-H, which has broad industry-wide relevance with a high degree of complexity. The experiments result shows our framework for type inference in SQL is useful for database comprehension. In a broader context, our work provides an instrumental technique in various applications, including reverse engineering, stored procedures maintenance and automatic schema matching.

Although our work focuses on SQL queries, the formalism introduced in this work, i.e. schema tableaux, can be applied to other SQL statements such as `Create` and `Insert` due to the similarity of SQL statements. This is a potential extension that we would like to explore in a future work.

1.4 Outline of the Thesis

The thesis is organised as follows.

Chapter 2 presents a survey on the related work on type inference in database programming. Especially, we review the formalism of type formula in relational algebra context proposed by Bussche and Waller [4]. The principal type of an relational algebra expression specifies all assignments of types, namely sets of attributes, to relation names, under which a given relational algebra expression is well-typed, as well as the output type that expression will have under each of these assignment.

When we move from relational algebra to SQL expression, the situation is very different. Issues such as data type and column position of attributes in SQL are missing in relational algebra. If we try to translate SQL expression to the primitive form of relation algebra, such type information are lost. Chapter 3 takes up the SQL specific features and formally defines schema, and develops a grammar that includes the core features of the standard SQL. Moreover, properties are assigned to the syntactic rules by semantic rules. On the other hand, a new formalism called schema tableau is associated with nodes of the derivation trees corresponding to query generated by the grammar. We show schemas satisfying nodes of the derivation trees can be represented by schema tableaux associated with nodes of the derivation trees. Furthermore, we introduce an important operator *merge* and rules to compute schema tableaux. Issues such as unambiguity and non-circularity of the grammar are discussed.

Chapter 4 manifests how the proposed framework works in practice. We introduce an architecture of a system that takes as input a collection of SQL expression and generates schema tableaux of the database. The system consists of a parser, and type inference engine, and a conjugator. Then we apply the system on the suite of queries of TPC-H. While the TPC-H is a benchmark for performance, the suite of queries are also good test input for our case that represents the real world situations, due to the broad industry-wide relevance with a high degree of complexity they have. The experiments indicate the proposed framework is very useful in the context of database schema comprehension.

We draw conclusion in Chapter 5. We also identify possible directions of future work.

Chapter 2

Background Review

This chapter gives a general account on related work, followed by a review of type inference in relational algebra.

2.1 Related Work

In their work on the language Machiavelli [11, 3], Ohori, Buneman and Breazu-Tannen were probably the first to introduce type inference in the context of database programming languages. Machiavelli features polymorphic field selection from nested records, as well as a polymorphic join operator. However, the inference of principal types for full-fledged relational algebra expressions was not taken up in that work. We should also mention the work of Stemple et al. [14], who investigated reflective implementations of the polymorphic relational algebra operators.

Other important related work is that on the extension of functional programming languages with polymorphic record types. Some of the most sophisticated proposals in that direction were made by Rémy [12, 13]. This work adds record types to the type system of ML, featuring polymorphic field selection and record concatenation. While this system captures many realistic functional programs involving records, it cannot express the conditions on the types of relations implied by certain relational algebra expressions, such as

$$\delta_{A<5}(r \bowtie s) \bowtie ((r \times u) - v).$$

Notably constraints such as set disjointness (needed for the operator $\times$) or set equality (for the operator $\cup$), cannot be expressed in these systems. The reason is probably the additional concern of these systems for subtyping: a program applicable to records of a certain type should more generally be applicable to records having all the fields of that type and possibly more [4]. This is clearly not true for relational algebra expressions.

Bussche and Waller gave a polymorphic account of the relation algebra. In their paper [4], they introduced a formalism of type formula specifically for relational algebra expressions. They also presented an algorithm that computes the principal type for a given relational algebra expression. The principal type of an expression is a formula that specifies all assignments of types, namely sets of attributes, to relation names, under which a given relational algebra expression is well-typed , as well as the output type that expression will have under each of these assignments.

In the paper [4], the concept of a polymorphic query was defined. Topics such as complexity, the relationship with monadic logic, and polymorphic expressive power were also discussed.

2.2 Type Inference in the Polymorphic Relational Algebra

This section will briefly review the major concepts of the type formula for relational algebra.

Schemas, Types, and Expressions

A schema is a finite set S of relation variables. A type is a finite set τ of attribute names. Let S be a schema. A type assignment on S is a mapping $\mathcal{T}$ on S, assigning to each $r \in S$ a type $\mathcal{T}(r)$. So, the usual notion of database schema, which specifies both the relation names and the associated sets of attributes, splits in two notions.

The expressions of the relational algebra are defined by the following grammar:

$$e \rightarrow r \mid (e \cup e) \mid (e - e) \mid (e \bowtie e) \mid (e \times e) \mid \sigma_{\theta(A_1,\ldots,A_n)}(e) \mid \pi_{A_1,\ldots,An}(e) \mid \rho_{A/B}(e) \mid \hat{\pi}_A(e)$$

Here e denotes an expression, r denotes a relation variable, and A, B, and A_i denote attribute names. And θ, π and $\hat{\pi}$ refer to a selection, projection and project out respectively.

Well-typed Expressions and Typable Expressions

Let S be a schema , e an expression with $Relvars(e) \subseteq S$, where $Relvars(e)$ denotes all relation variables in e, $\mathcal{T}$ a type assignment on S, and τ a type. The rules for when e has type τ given $\mathcal{T}$, denoted by $\mathcal{T} \vdash e : \tau$, are given in Figure 1.

Definition Let e be an expression and let $\mathcal{T}$ be a type assignment on $\mathit{Relvars}(e)$. If there exists a type τ such that $\mathcal{T} \vdash e : \tau$, we say that e is well-typed under $\mathcal{T}$.

Expression e is called typable if there exists a type assignment $\mathcal{T}$ on $\mathit{Relvars}(e)$ such that e is well-typed under $\mathcal{T}$.

Type Contexts and Type Formulas

Consider the expression

$$e = \sigma_{B=C}(\rho_{A/B}(r) \cup s) \bowtie u.$$

This expression is well-typed under exactly those type assignments $\mathcal{T}$ satisfying the following two conditions:

1. $\mathcal{T}(s) = (\mathcal{T}(r) - \{A\}) \cup \{B\}$;
2. C must belong to at least one of $\mathcal{T}(s)$, $\mathcal{T}(t)$ or $\mathcal{T}(u)$.

Given such $\mathcal{T}$, the type of e then will equal $\mathcal{T}(s) \cup \mathcal{T}(u)$.

All the above information is expressed by the following type formula for e:

$$\begin{array}{lcl}
r : a_1a_2 & & \\
s : a_1a_2 & \leadsto & e : a_1a_2a_3 \\
u : a_2a_3 & & \\
A : r \wedge \neg s & & A : u \\
B : s \wedge \neg r & & B : \mathit{true} \\
C : (r \leftrightarrow s) \wedge (r \vee s \vee u) & & C : \mathit{true}
\end{array}$$

$$\frac{\mathcal{T}(r) = \tau}{\mathcal{T} \vdash r : \tau}$$

$$\frac{\mathcal{T} \vdash e_1 : \tau \qquad \mathcal{T} \vdash e_1 : \tau}{\mathcal{T} \vdash (e_1 \cup e_2) : \tau}$$

$$\frac{\mathcal{T} \vdash e_1 : \tau \qquad \mathcal{T} \vdash e_1 : \tau}{\mathcal{T} \vdash (e_1 - e_2) : \tau}$$

$$\frac{\mathcal{T} \vdash e_1 : \tau_1 \qquad \mathcal{T} \vdash e_1 : \tau_2}{\mathcal{T} \vdash (e_1 \bowtie e_2) : \tau_1 \cup \tau_2}$$

$$\frac{\mathcal{T} \vdash e_1 : \tau_1 \qquad \mathcal{T} \vdash e_1 : \tau_2 \qquad \tau_1 \cap \tau_2 = \emptyset}{\mathcal{T} \vdash (e_1 \times e_2) : \tau_1 \cup \tau_2}$$

$$\frac{\mathcal{T} \vdash e : \tau \qquad A_1, ..., A_n \in \tau}{\mathcal{T} \vdash \sigma_{\theta(A_1,...,A_n)}(e) : \tau}$$

$$\frac{\mathcal{T} \vdash e : \tau \qquad A_1, ..., A_n \in \tau}{\mathcal{T} \vdash \pi_{(A_1,...,A_n)}(e) : \{A_1, ..., A_n\}}$$

$$\frac{\mathcal{T} \vdash e : \tau \qquad A \in \tau \qquad B \notin \tau}{\mathcal{T} \vdash \rho_{A/B}(e) : (r - \{A\}) \cup \{B\}}$$

$$\frac{\mathcal{T} \vdash e : \tau \qquad A \in \tau}{\mathcal{T} \vdash \hat{\pi}_A(e) : r - \{A\}}$$

Figure 1: Typing Rules of the Relational Algebra

The left-hand side of the formula, called type context, along with the right-hand side of the formula, which specifies relational algebra e, output variables and output attribute, depict the condition under which the relational algebra expression is well-typed and the output of it. This type formula can be read as follows. Expression e is well-typed under precisely all type assignments that can be produced by the following procedure:

1. Instantiate a_1, a_2, and a_3 by any three types, on condition that they are pairwise disjoint, and do not contain A, nor B, nor C.

2. Preliminarily assign type $a_1 \cup a_2$ to r; $a_1 \cup a_2$ to s; and $a_2 \cup a_3$ to u.

3. In this preliminary type assignment, A must be added to the type of r, but must not be added to that of s; where it is added to the type of u is a free choice.

4. Similarly,B must be added to the type of s, not to that of r, and freely to that of s.

5. Finally, C must be added at least to one of the types of r, s, and u, but if we add it to r we must also add it to s and vice versa.

The type of e under a type assignment thus produced equals $a_1 \cup a_2 \cup a_3$, to which we must add B and C, and to which we also add A on condition that it belongs to the type of u.

2.3 Solving Systems of Set Equations

Type inference algorithm for programming languages typically work by structured induction on program expressions, enforcing the typing rules "in reverse", and using some form of unification to combine type formulas of subexpressions. In the context of relational algebra, relation types are sets, so we need a replacement for classical unification on terms. This role will be played by the following algorithm for solving systems of set equations.

Symbolic Solutions

Fix some universe $\mathcal{U}$. In principle $\mathcal{U}$ can be any set, but in our intended application $\mathcal{U}$ is the universe of attribute names. Assume further given a sufficiently large supply of variables. In our intended application, this role will be played by type variables. An equation is an expression of the form $lhs = rhs$, where both lhs and rhs are sets of variables. A system of equations, such that every variable occurring at the left-hand side of some equation is in L, and every variable at right-hand side of some equation is in R.

A substitution on a set S of variables is a mapping from S to the subsets of $\mathcal{U}$. A substitution is called proper if different variables are assigned disjoint sets. A valuation of a system Σ consists of a proper substitution on L and a proper substitution on R. A valuation (f_L, f_R) is a solution of Σ if for every equation

$$a_1 \dots a_m = b_1 \dots b_n$$

in Σ, we have

$$f_L(a_1) \cup \ldots \cup f_L(a_m) = f_R(b_1) \cup \ldots \cup f_R(b_n).$$

A symbolic valuation of Σ consists of a new set V of variables and a mapping g from $L \cup R$ to the subsets of V. Take some proper substitution h on V. Now define the following substitution h_L on L: for any $a \in L$,

$$h_L(a) := \bigcup \{h(c) | c \in g(a).\}$$

In a completely analogous way the substitution h_R on R is defined too. A symbolic valuation is called a symbolic solution of Σ if for every proper substitution h on V, the pair (h_L, h_R) is a solution of Σ, and conversely, every solution of Σ can be written in this way. So, a symbolic solution is a finite representation of the set of all solution.

Every system of equations Σ has a symbolic solution, which can be computed from Σ in polynomial time.

An Equation-solving Example

The following is an equation-solving example.

Consider Σ with $L = \{a_1, a_2, a_3\}$, $R = \{b_1, b_2, b_3\}$, and the equations $a_1 = b_1$ and $a_2 = b_1 b_2$.

From the first equation we deduce that

$$\overline{a}_1, (\overline{a}_1, \overline{b}_2), (\overline{a}_1, \overline{b}_3)$$

as well as

$$\overline{b}_1, (\overline{a}_2, \overline{b}_1), (\overline{a}_3, \overline{b}_1)$$

are also in V_0. So

$$V - V_0 = \{\bar{a}_1, \bar{b}_3, (\bar{a}_2, \bar{b}_2), (\bar{a}_3, \bar{b}_3),$$

and the symbolic solution g' is given by

$g'(a_1) = \emptyset \qquad g'(b_1) = \emptyset$

$g'(a_2) = (\bar{a}_2, \bar{b}_2) \qquad g'(b_2) = (\bar{a}_3, \bar{b}_3)$

$g'(a_3) = a_3, (\bar{a}_3, \bar{b}_3) \qquad g'(b_3) = \bar{b}_3, (\bar{a}_3, \bar{b}_3)$

If we rename the variables for added clarity, we obtain the symbolic solution

$a_1 = \emptyset \qquad b_1 = \emptyset$

$a_2 = c_1 \qquad b_2 = c_1$

$a_3 = c_2c_3 \qquad b_3 = c_3c_4$

which can be interpreted as specifying that the only solutions to Σ are those were we assign the same set to a_2 and b_2, which is disjoint from the sets assigned to a_3 and b_3 (the latter two sets need not be disjoint), and where a_1 and b_1 are empty.

2.4 From Relational Algebra to SQL

Considering the difference between relational algebra and SQL, observant readers may wonder how this formalism and algorithm can be applied to SQL. As we have noticed, the type on the level of individual attributes, which are almost present, is ignored in the work of Bussche and Waller [4]. For example, as they further explained, for $\sigma_{A=''John''}(r)$ to be well-typed it suffices for the algorithm that the type of r has an A-attribute. However, in reality, A must in addition be of type string. Incorporating types on the attribute value level only has an effect on the special attributes of an expression, and it has no effect on its polymorphic basis, claimed Bussche and Waller. It is true for this example. But L does need extra care to apply type inference in SQL, if we take other situations into account. Consider the following example:

```
Select A, B From R Union Select * From S.
```

In relational algebra, A and B must be attributes of s, but this is not necessary for SQL.

There is another issue that is important in SQL. In practice, the position of column plays an role in SQL. However, since relational algebra is a set notion, order does not matter.

There is also type information readily to be collected in SQL. The statement,

```
Select R.A From R, S,
```

implies relation R contain attribute A.

Furthermore, operators not captured by primitive form of relational algebra, such as aggregation, provide useful attribute and type information, too.

In light of the difference between relational algebra and SQL, we believe it is a natural approach to collect and infer attribute constraints and type information from SQL expressions directly, as the rest of this thesis will demonstrate.

Chapter 3

Proposal

Our major proposal is put forward in this chapter. We will have a closer look at issues such as data type and column position of attributes in SQL, which are missing in relational algebra. We take up these SQL specific features and propose a framework for type inference in SQL. Schema is formally defined. We consider a grammar G_{SQL} that includes the core features of the standard SQL. Furthermore, attributes is assigned to syntactic rules by a set of semantic rules. On the other hand, a new formalism called schema tableau is introduced. Nodes of the derivation trees are associated with schema tableaux. Then we devise rules to computed schema tableaux corresponding nodes of the derivation trees. We show that schema satisfying nodes of the derivation trees can be represented by schema tableaux obtained by our algorithm. Many aspects are discussed.

3.1 Schema

Definition Schema

Let *Rels* be a henceforth fixed finite set of *relation names*, and *Attrs* a henceforth fixed finite set of *attribute names.* Relation names will be denoted $R, S, U, \ldots$, possibly subscripted, and attribute names will be denoted $A, B, C, \ldots$. We assume that *Rels* contains two special relation names $\top$, and $\perp$. The name $\top$ refers to the (unique) relation whose arity is 0. The name $\perp$ refers to an relation whose name is not explicitly mentioned, such as the output relation of a query.

Furthermore, let *Types* be a henceforth fixed finite set of *Typenames.* Typenames will be denoted $\tau_1, \tau_2, \ldots$.

A *schema* is a mapping

$$\sigma : \mathit{Rels} \times \mathit{Attrs} \times \mathbb{N} \longrightarrow \mathit{Types},$$

such that for each relation $R \in \mathit{Rels}$, there is a subset $\mathbf{A}$ of Attrs of cardinality k, for some $k \in \mathbb{N}$, such that $\sigma(R, A, m)$ is defined if and only if $A \in \mathbf{A}$, and $1 \leq m \leq k$, and furthermore, each attribute participates at most once in $\sigma(R, \cdot, \cdot)$.

While attribute names and relation names can be of any length in theory, once given, however, its length is fixed. In practice, the range of length allowed for attribute names and relation names is specified by DBMS vendors. Char in DB2, for example, can have 1 to 254 bytes fixed length, while char in Oracle has maximum length of 2000 bytes. Hence, it does not lose generality by assuming attribute names and relation names are finite sets.

Note that there may be many unnamed relations, and each occurrence of $\perp$ may refer to a different one. We have this notion such the formalism is more uniform.

Example: The schema

```
CREATE TABLE r (
 A int
 B char
);
CREATE TABLE s (
 C char
 D char
);
```

corresponds to

$$\sigma = \{(R, A, 1) \rightarrow \mathit{int}, (R, B, 2) \rightarrow \mathit{char}, (S, C, 1) \rightarrow \mathit{int}, (S, D, 2) \rightarrow \mathit{char}\}$$

For brevity, we shall usually write

$$\sigma = \{R\langle A\colon \mathit{int}, B\colon \mathit{char}\rangle, S\langle C\colon \mathit{char}, D\colon \mathit{char}\rangle\}.$$

Grammar of SQL Query

Since there exists a unique derivation tree for each leftmost derivation of a string, and vice versa [8], we shall move back and forth freely between the two.

Consider query w,

`Select A, C From R, S Where B = D.`

we have the derivation sequence,

`<SAW>` $\Longrightarrow$ `Select <SL> From <FL> Where <Cond>` $\overset{*}{\Longrightarrow} w$.

We want to know what is meant by node `<FL>`. Meaning conveyed by the expression `<FL>` includes relations R and S, and, attributes occurring in R, and S. Furthermore, our interest is those databases that satisfies such a node. Schema $\{R(A, B), S(C, D)\}$, for example, is one that satisfies node `<FL>`, while schema $\{U(A, B)\}$ is not.

Schemas satisfying a node in a derivation tree is decided by what relation and attribute names are associated to the node. At times, it appears convenient to have this redundancy. Therefore, we consider it part of the 'meaning' of the nodes too.

In other to appreciate the semantics of SQL expressions, we assign *attributes* to production rules, called *syntactic rules*, by another set of rules called *semantic rules* [9]. The reader should distinguish the term attribute in this context from one occurring in tables as column name.

We define a mapping m as follows:

$$m : \Sigma \rightarrow \{0, 1\}$$

where Σ is a set of schemas.

Three type of attributes are associated to the grammar of SQL:

1. $rels(\nu)$;

2. $attrs(\nu)$;

3. $schemas(\nu)$.

An *SQL query* is a string in the following grammar:

1. `<SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>`

   $rels(<\texttt{SFW}>) = rels(<\texttt{SL}>) \cup rels(<\texttt{FL}>) \cup rels(<\texttt{Cond}>)$

   $attrs(<\texttt{SFW}>) = attrs(<\texttt{SL}>) \cup attrs(<\texttt{FL}>) \cup attrs(<\texttt{Cond}>)$

2. `<SelList> ::= <Attribute>, <SelList>`

   $rels(<\texttt{SL}>) = rels(<\texttt{Attr}>) \cup rels(<\texttt{SL}>)$

   $attrs(<\texttt{SL}>) = attrs(<\texttt{Attr}>) \cup attrs(<\texttt{SL}>)$

3. `<SelList> ::= <Attribute>`

   $rels(<\texttt{SL}>) = rels(<\texttt{Attr}>)$

   $attrs(<\texttt{FL}>) = attrs(<\texttt{Attr}>)$

4. `<Attribute> ::=` $A_1|\ldots|A_i|B_1|\ldots|B_j|C_1|\ldots|C_k$

   $rels(<\texttt{Attr}>) = \emptyset$

   $attrs(<\texttt{Attr}>) = A_1|\ldots|A_i|B_1|\ldots|B_j|C_1|\ldots|C_k$

5. `<FromList> ::= <Relation>, <FromList>`

   $rels(<\texttt{FL}>) = rels(<\texttt{Rel}>) \cup rels(<\texttt{FL}>)$

   $attrs(<\texttt{FL}>) = attrs(<\texttt{Rel}>) \cup attrs(<\texttt{FL}>)$

6. `<FromList> ::= <Relation>`

   $rels(<\texttt{FL}>) = rels(<\texttt{Rel}>)$

   $attrs(<\texttt{FL}>) = attrs(<\texttt{Rel}>)$

7. `<Relation> ::=` $R_1|\ldots|R_i|S_1|\ldots|S_j|U_1|\ldots|U_k$

$rels(<\texttt{Rel}>) = R_1|\ldots|R_i|S_1|\ldots|S_j|U_1|\ldots|U_k.$

$attrs(<\texttt{Rel}>) = \emptyset.$

8. `<Condition> ::= <Attribute> = <Attribute>`

$rels(<\texttt{Cond}>) = rels(<\texttt{Attr}>) \cup rels(<\texttt{Attr}>)$

$attrs(<\texttt{Cond}>) = attrs(<\texttt{Attr}>) \cup attrs(<\texttt{Attr}>)$

9. `<Query> ::= <SFW> Union <SFW>`

$rels(<\texttt{Cond}>) = rels(<\texttt{Attr}>) \cup rels(<\texttt{Attr}>)$

$attrs(<\texttt{Cond}>) = attrs(<\texttt{Attr}>) \cup attrs(<\texttt{Attr}>)$

Unambiguity of the Grammar

Theorem 1 *Grammar G_{SQL} is unambiguous.*

Proof: We will prove the result in two steps. First we show the set of rules without `Union` is unambiguous; then we extend the set of rules to include `Union`, and prove its unambiguity. In either case, the start symbol is `<Query>`.

Step 1:

Consider the set of rules without `Union`.

We will prove the theorem by induction.

Basic:

The length of shortest of query strings generated by grammar G_{SQL} is 8.

`<Query>`$\Longrightarrow$

```
<SFW>⟹

Select <SL> From <FL> Where Condition ⟹

Select <Attribute> From <FL> Where Condition ⟹

Select A From <FL> Where Condition ⟹

Select A From <Relation> Where Condition ⟹

Select A From R Where Condition ⟹

Select A From R Where <Attribute> = <Attribute> ⟹

Select A From R Where B = <Attribute> ⟹

Select A From R Where B = C
```

where A, B, C are arbitrary terminals as attribute names, and R arbitrary terminal as relation name. There is only one leftmost derivation for query string w, where $|w| = 8$.

Induction: Assume for query string w, where $|w| = n \geq 8$, there is only one leftmost derivation. We observe variable <Cond> expands to a fixed length, but variables <SL> and <FL> can expand to various length. Moreover, the length of query strings increase by 2 for an additional attribute in the <SL>, for example, ', B', or for an additional relation in <FL>. Consider now query string w', where $|w'| = n + 2$, as follows:

Select $A_1, \ldots, A_i$, From $R_1, \ldots, R_j$ Where <B> = <C>.

where $i, j \geq 1$.

If $i \geq 2$, there exists a query string w, where $|w| = n$, as follows:

Select $A_1, \ldots, A_{i-1}$ From $R_1, \ldots, R_j$ Where <B> = <C>.

By our assumption, there exists a unique derivation for string w, as follows:

`<Query>`$\Longrightarrow$ `<SFW>`$\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_{i-1}$ `From <FL> Where <Condition>` $\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_{i-1}$ `From` $R_1, \ldots, R_j$ `Where <Condition>` $\overset{*}{\Longrightarrow}$ w.

Hence, we can obtain a unique leftmost derivation for string w':

`<Query>`$\Longrightarrow$ `<SFW>`$\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_{i-1}$, `<SL> From <FL> Where <Condition>` $\Longrightarrow$ `Select` $A_1, \ldots, A_{i-1}$, `<SL> From <FL> Where <Condition>` $\Longrightarrow$ `Select` $A_1, \ldots,$ `<Attribute> From <FL> Where <Condition>`$\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_i$ `From` $R_1, \ldots, R_j$ `Where <Condition>` $\overset{*}{\Longrightarrow}$ w'.

If $i = 1$, then $j \geq 2$, since $|w'| = n + 2 \geq 10$. There exists a query string w, where $|w| = n$, as follows:

`Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_{j-1}$ `Where <B> = <C>`.

By our assumption, we have a unique leftmost derivation for w as follows:

`<Query>`$\Longrightarrow$ `<SFW>`$\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_i$ `From` $R_1, \ldots, R_{j-1}$ `Where <Condition>` $\overset{*}{\Longrightarrow}$ w.

Hence, we obtain a unique leftmost derivation for string w'.

`<Query>`$\Longrightarrow$ `<SFW>`$\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_i$ `From` $R_1, \ldots, R_{j-1}$, `<FL> Where <Condition>`$\Longrightarrow$ `Select` $A_1, \ldots, A_i$ `From` $R_1, \ldots, R_{j-1}$, `<Relation> Where <Condition>` $\Longrightarrow$ `Select` $A_1, \ldots, A_i$ `From` $R_1, \ldots, R_j$ `Where <Condition>` $\overset{*}{\Longrightarrow}$ w'.

We showed query strings generated by G_{SQL} has only one leftmost derivation.

Step 2:

We extend the rules to include `Union`.

Let w be a query string generated by grammar G_{SQL}.

If no `Union` is involved, then w is in the following form:

`Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_j$ `Where <B> = <C>.`

This is exactly the case of step 1. We are done.

If `Union` is involved, then w is in the following form:

`Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_j$ `Where <C> = <D> Union Select` $B_1, \ldots, B_i$, `From` $S_1, \ldots, S_j$ `Where <E> = <F>.`

We have

`<Query>` $\Longrightarrow$ `<SFW> Union <SFW>.`

By step 1, we have a unique leftmost derivation for variable `<SFW>`. Hence, we have unique leftmost derivation ,

`<Query>` $\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_j$ `Where <C> = <D> Union <SFW>` $\overset{*}{\Longrightarrow}$ `Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_j$ `Where <C> = <D> Union Select` $B_1, \ldots, B_i$, `From` $S_1, \ldots, S_j$ `Where <E> = <F>.`

We showed the whole set of rules is unambiguous. Proved.

Testing for Circularity

Now we need to show that the collection of semantic rules, as described previously, is well defined. Note attribute $schemas(\nu)$ of a node in derivation trees is decided by attributes $rels(\nu)$ and $attr(\nu)$ of the node only as discussed earlier, hence, attribute $schema(\nu)$ does not introduce extra degree of complexity in terms of deciding the circularity. For brevity, we omit its presence in the directed graph.

Theorem 2 *The collection of semantic rules of grammar G_{SQL} is non-circular.*

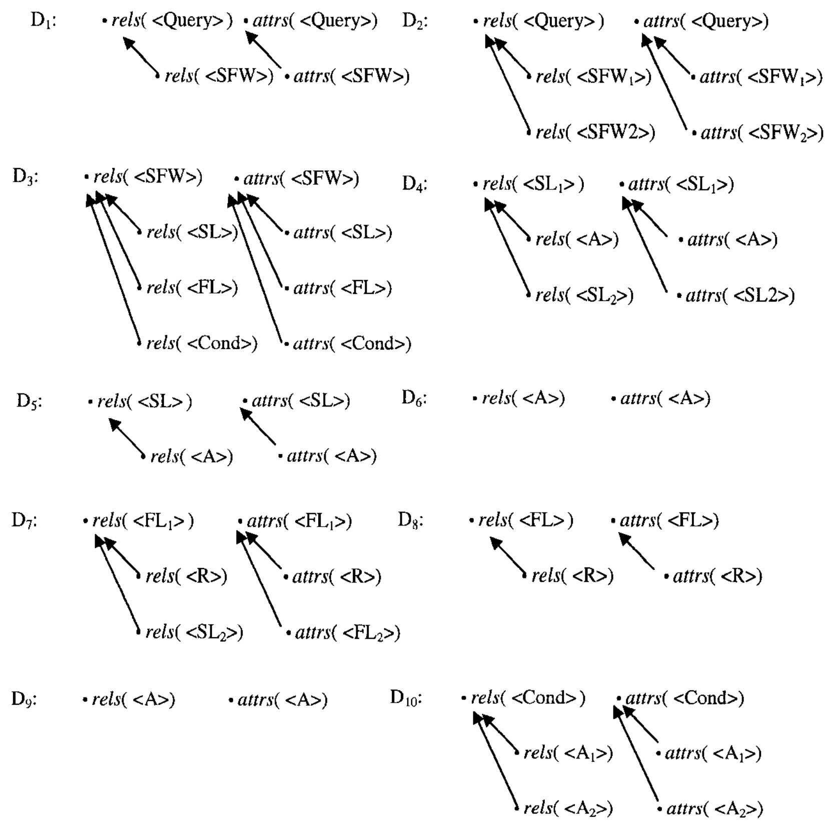

D1: • rels(<Query>) • attrs(<Query>)
rels(<SFW>) • attrs(<SFW>)
D2: • rels(<Query>) • attrs(<Query>)
rels(<SFW1>) • attrs(<SFW1>)
rels(<SFW2>) • attrs(<SFW2>)
D3: • rels(<SFW>) • attrs(<SFW>)
rels(<SL>) • attrs(<SL>)
rels(<FL>) • attrs(<FL>)
rels(<Cond>) • attrs(<Cond>)
D4: • rels(<SL1>) • attrs(<SL1>)
rels(<A>) • attrs(<A>)
rels(<SL2>) • attrs(<SL2>)
D5: • rels(<SL>) • attrs(<SL>)
rels(<A>) • attrs(<A>)
D6: • rels(<A>) • attrs(<A>)
D7: • rels(<FL1>) • attrs(<FL1>)
rels(<R>) • attrs(<R>)
rels(<SL2>) • attrs(<FL2>)
D8: • rels(<FL>) • attrs(<FL>)
rels(<R>) • attrs(<R>)
D9: • rels(<A>) • attrs(<A>)
D10: • rels(<Cond>) • attrs(<Cond>)
rels(<A1>) • attrs(<A1>)
rels(<A2>) • attrs(<A2>)

Figure 2: Directed Graphs of the Rules

Proof: We will prove it in two steps. First we show the semantic rules without `Union` is non-circular; then we extend the rules to include `Union`, and prove its non-circularity.

According to the algorithm [9], we have a collection of directed graphs, shown in Figure 2, that corresponds to the semantic rules. Then we can build a direct graph for each derivation tree by "pasting together" various subgraphs.

Step 1:

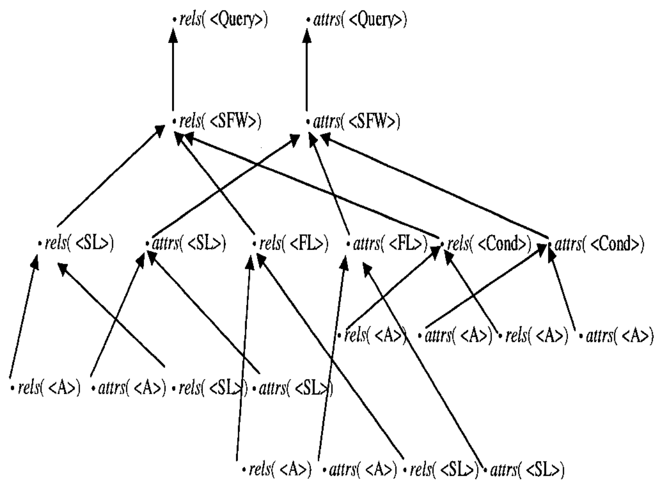


Figure 3: Non Circularity of Semantic Rules of G_{SQL}

We will prove it by induction. Let $D(w)$ be the directed graph corresponding to the derivation tree of string w.

Basic: The basic case is query string of length of 8, as follows:

`Select A From R Where B = C`

Its corresponding directed graph is shown in Figure 3. Obviously, there is no oriented circle.

Induction: Assume $D(w)$ is non-circular, when $|w| = n \geq 8$. As we discussed earlier, the length of strings grows by 2, for an additional attribute in `<SL>`, or an additional relation in `<FL>`. Given a string w', where $|w'| = n + 2$, as follows:

`Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_j$ `Where <Condition>.`

where $i, j \geq 1$.

If $i \geq 2$, there exist a query string w, where $|w| = n$, as follows:

`Select` $A_1, \ldots, A_{i-1}$ `From`$R_1, \ldots, R_j$ `Where <Condition>.`

Since $D(w)$ is non-circular as we assumed, we can construct $D(w')$ by adding the subgraph of D_3 to $D(w)$. which obviously does not result in oriented circle. So, $D(w')$ is non-circular too.

If $i = 1$, then $j \geq 2$, since $|w'| = n + 2 \geq 8$. There exist a query string w, where $|w| = n$, as follows:

`Select` $A_1, \ldots, A_i$ `From`$R_1, \ldots, R_{j-1}$ `Where <Condition>.`

Since $D(w)$ is non-circular by our assumption, we can construct $D(w')$ by adding the subgraph of D_5 to $D(w)$, which obviously does not result in oriented circle. So, $D(w')$is non-circular too.

Step 2:

We now extend the rules to include `Union`. Consider string w generated by grammar G_{SQL}.

If no `Union` is involved in string w. This is exactly the case in steps 1. We are done.

If `Union` occurs in w, it is in the following form:

`Select` $A_1, \ldots, A_i$, `From` $R_1, \ldots, R_j$ `Where <C> = <D> Union Select` $B_1, \ldots, B_i$, `From` $S_1, \ldots, S_j$ `Where <E> = <F>.`

We have a unique leftmost derivation,

`<Query>`$\overset{*}{\Longrightarrow}$`<SFW> Union <SFW>`$\overset{*}{\Longrightarrow} w$.

By step 1, the directed graphs corresponding to derivation trees whose root are labelled `<SFW>` is non-circular. So, we have two subgraphs for derivation tree whose roots are labelled `<SFW>`. We can construct the directed graph $D(w)$ by "pasting together" the two subgraphs and subgraph D_2. Obviously, this does not result in any oriented circle. We showed the semantic rules with `Union` is non-circular.

Proved.

Well-typed Query

Let q be an SQL query generated by the grammar above, and let σ be a schema. We say that the query q is well-typed wrt σ, denoted

$$\sigma \models q$$

if the parse tree Θ_q of q has the following properties:

1. For ν labelled `<SFW>`, let $\nu_1, \ldots, \nu_6$ be the children of ν, from left to right. We have $\sigma \models \nu$ if for all $A \in attrs(\nu_2)$, there is an $R \in rels(\nu_4)$, and an $i > 0$, such that $\sigma(R, A, i)$ is defined. Furthermore, for all $A \in attrs(\nu_6)$, there is an $R \in rels(\nu_4)$, and an $i > 0$, such that $\sigma(R, A, i)$ is defined.

2. For ν labelled `<Cond>`, let ν_1, ν_2, ν_3 be the children of ν, from left to right, and let $attrs(\nu_1) = \{A\}$, and $attrs(\nu_3) = \{B\}$. Then we require that there exist $\{R, S\} \subset rels(\nu_4)$, and $i, j > 0$, such that $\sigma(R, A, i) = \sigma(S, B, j)$.

There are two fundamental issues in typing, that is, type checking and type inference.

The *type checking problem* is, given a query q and a schema δ, to decide whether q is well typed wrt δ. This is done routinely in every database system, for every query, but not always that well, as we shall see later.

The *type inference problem* is, given a query q and its parse tree Θ_q with root ν , to determine $schemas(\nu)$.

3.2 Schema Tableaux

Researchers showed conditional table is a powerful representation form of incomplete database [1, 6]. Potentially, all information is not available in database schema comprehension. Hence, a c-table-like structure will be desired as we will introduce next.

Assume countably infinite sets $Attrvars = \{X, Y, \ldots\}$, and $Typevars = \{x, y, \ldots\}$.

An expression such as $R\langle A\colon int, B\colon char\rangle$ is called schema atom. A schema atom can take one of several forms.

Definition Schema Atom

A *ground schema atom* is an expression of the form

$$R\langle A_1\colon \tau_1, \ldots, A_n\colon \tau_n\rangle$$

where $R \in Rels$, $\tau_i \in Types$, and $A_i \in Attrs$, for $i \in \{1, \ldots, n\}$.

The position of each attribute in the table to which it belongs is reflected by this sequence notion of the schema atom.

A schema atom is called *non-ground schema atom*, if some attribute names are replaced by attribute name variables from *Attrvars*, and/or some type names are replaced by type variables from *Typevars*.

A schema atom can also be of one of the following forms:

$$R\{A_1 : \tau_1, \ldots, A_n : \tau_n\}$$

Intuitively, such an atom means that the schema of R contains attributes A_i of type τ_i, for $i \in \{1, \ldots, n\}$, but the order of the attributes are unknown.

$$R\{A_1 : x_1, \ldots, A_n : x_n\}*$$

The meaning is similar to the previous atom, except in this case there might be more attributes than the A_i's in the schema of relation R.

When convenient, we will use the underscore character '_' as an anonymous attribute name or type name variable. Not that each occurrence of '_' stands for a different variable.

Example: We have already seen schema atoms of the forms $R\langle A : int, B : char\rangle$, $R\{A : int, B : char\}$, $R\{A : int, B : char\}*$, and $R\{A : x, B : x\}$. The first atom means that table R has exactly attribute A of type *int* in its first column, and B of type *char* the second column.

If we know attributes A and B are in relation R, but we do not know their data types, or relative order, we write $R\{A : \_, B : \_\}$.

The output of a query is usually an unnamed relation, which we denote $\perp\langle A : int, B : char\rangle$, for instance.

Schema atoms are denoted $\alpha, \beta, \ldots$.

Let α be a schema atom. By $rel(\alpha)$ we mean the relation name occurring in α. For example: $rel(S\{A : int\}*) = S$, and $rel(R\langle\langle A : int, B : char\rangle\rangle) = R$.

A *condition box* is a finite set of *attribute constraints* of the form

$$(A : \tau) : \phi$$

where A is an attribute name, τ a data type, and ϕ a Boolean formula with atoms from *Rels*.

For instance,

$$(A : int) : (R \vee S) \wedge U$$

is a constraint saying attribute A is of type *int*, and A is in either relation R or S, but not in relation U.

Definition AD Pair

A pair consisting of an attribute name or variable, and data type or variable, is an important component used in our discussion. For brevity, we call such a pair an *AD pair*. An AD pair is denoted $(X : x)$. Its instance could be of one of the following forms:

- $(X : x)$, where both X and x are variables;
- $(A : x)$, where A is a concrete attribute name, while x is data type variables;
- $(X : \tau)$, where X is a attribute name variable, while τ is a concrete data type;
- $(A : \tau)$, where A is a concrete attribute name, and τ is a concrete data type.

A *schema tableau* T is a finite set of schema atoms , each over a different relation name, and a condition box, denoted T_Φ.

Example:

$T = \{R\langle X : x, Y : y, Z : z\rangle, S\{A : x, B : y, C : z\} *\},$

$T_\Phi = \{(D : \_) : R \vee S, (E : \_) : R \vee S\}.$

While schemas tableaux are represented concisely by expressions as seen earlier, they might be displayed in tableaux too. There is no semantic difference between the two forms of representation, except the form of a tableau may be more visual.

The following is the tableau form of the schema tableau for the same example.

		$(D : \_) : R \vee S$ $(E : \_) : R \vee S$
R	$\langle X : x, Y : y, Z : z\rangle$	
S	$\{A : x, B : y, C : z\}*$	

As we have seen, a schema tableau represents incomplete information about a schema, as it partially specifies the state of a schema. In formalising this notation we shall use the Open World Assumption (OWA) [14]. Instead of completely identifying one schema, a schema tableau is compatible with many schemas. A representation of a schema tableau is a possible schema. The set of schemas represented by a schema tableau T, denoted $rep(T)$, is defined as follows using the concept of valuations.

A *valuation* is a mapping from attribute variables to attributes, and type variables to types.

Given an atom $\alpha \in T$, we define $h(\alpha)$ as the following (homomorphic) extension of h:

1. If α is of the form $R\langle X_1 : x_1, \ldots, X_n : x_n\rangle$,

   then $h(\alpha) = R((h(X_1) : h(x_1), \ldots, h(X_n) : h(x_n))$.

2. If α is of the form $R\{X_1 : x_1, \ldots, X_n : x_n\}$,

   then $h(\alpha) = R(h(X_{i_1}) : h(x_{i_1}), \ldots, h(X_{i_n}) : h(x_{i_n}))$, for some non-deterministically chosen permutation $i_1, \ldots, i_n$ of $1, \ldots, n$.

3. If α is of the form $R\{X_1 : x_1, \ldots, X_n : x_m\}*$, then $h(\alpha) = R(A_1\!: \tau_1, \ldots, A_m\!: \tau_n)$ for some $n \geq m$, and $h(X_i\!: x_i) = (A_j\!: a_j)$, for some $j \in \{1, \ldots, n\}$.[1]

A valuation h *satisfies* a condition box, for each AD pair $(X\!: x)\!: \phi$, we have that $(h(X) : h(x))$ satisfies ϕ. Let ϕ be an atomic formula, say R. Then $(h(X) : h(x))$ satisfies R if $h(X) = R$. If ϕ is not atomic, we just apply standard propositional semantics.

We are now ready to define the set of schemas represented by a schema tableau:

$$rep(T) = \{h(T) : h \text{ is a valuation that satisfies } T_\Phi\}.$$

Example:

Assume T is a tableau, with $T_s = \{R\{A : int, B : int\}*, S\langle Y_1 : int, Y_2 : char\rangle\}$, and $T_\Phi = \{(X_1 : char) : R\}$. Since we follow the OWA, each of the following schemas, among others, is a member of $rep(T)$:

1. $\{R\{A : int, B : int, C : char\}, S\langle E : int, F : char\rangle\}$;

[1]Note that in a proper substitution each i will correspond to a unique j. See proper substitution.

2. $\{R\{A : int, B : int, C : char, E : char\}, S\langle F : int, G : char\rangle\}$;

3. $\{R\langle C : char, B : int, A : int\rangle, S\langle E : int, F : char\rangle\}$;

4. $\{R\langle C : char, B : int, A : int, F : int, G : char\rangle, S\langle H : int, I : char\rangle\}$;

However, both schemas $\{R\{A : int, B : int\}, S\langle E : int, F : char\rangle\}$ and $\{R\{A : int, C : char\}, S\langle E : int, F : char\rangle\}$ are not in *rep*(T), because condition box is not satisfied in the former, while (B : *int*) is missing from R in the latter. If a specified order of attributes in tableau T is not respected in a given schema S, S is not a member of *rep*(T), either. Schema $\{R\{A : int, B : int, C : char\}, S\langle E : char, F : int\rangle\}$ is such an example.

We shall also need substitutions. A *substitution* is a mapping from variables to variables and constants, which is extended to be the identity on constants and generalised to schema atoms and condition boxes in the natural fashion.

Definition Substitution of schema atom

Fix a schema σ. Let schema atom $\alpha = R\{X_1 : x_1, \ldots, X_n : x_n\}$, ϑ a substitution from the variables occurring in α to constants and/or variables in σ. We define

$$\alpha\vartheta = R\{X_1\vartheta : x_1\vartheta, \ldots, X_n\vartheta : x_n\vartheta\}$$

Example: Assume $\alpha = R\{X_1 : x_1, B : x_2, X_3 : x_1\}$ and $\vartheta = \{x_1/int, X_1/A\}$, then

$$\alpha\vartheta = R\{A : int, B : x_2, X_3 : int\}$$

Definition Substitution of schema tableau

Fix a schema σ. Assume $T(\nu) = \{\alpha_1, \ldots, \alpha_n\}$ for some $n \geq 0$, where α_i is a schema atom. Let ϑ be a substitution from the variables occurring in $T(\nu)$ to constants and/or variables in σ. We define

$$T(\nu)\vartheta = \{\alpha_1\vartheta, \ldots, \alpha_i\vartheta\}$$

We observe $*$ can be "substituted" to one or more AD pairs.

Example: Suppose schema atom $\alpha = R\{X : \mathit{int}\}*$, each of the following, among others, is a possible state of the world described by α :

1. $R\langle A : \mathit{int}, B : \mathit{int}, C : \mathit{char}\rangle$;
2. $R\langle A : \mathit{int}, C : \mathit{int}\rangle$;
3. $R\langle A : \mathit{int}\rangle$.

Merging Schema Tableaux

Type inference algorithm calls for some form of unification. In our case, type is in the form of schema tableaux. We will introduce an operator that takes as input two schema tableaux, combines and unifies them, and generates a schema tableau.

Lemma 1 *Given schema tableaux T and T', there exists a schema tableau U such that $rep(U) = rep(T) \cap rep(T')$.*

Proof:

If $rep(T) = \emptyset$, or $rep(T') = \emptyset$, then $U = \emptyset$. We are done.

Now assume $rep(T_1) \neq \emptyset$ and $rep(T_2) \neq \emptyset$. We construct U such that U_s has exactly the schema atoms in T_s and T'_s , and U_Φ has exactly the AD pairs in T_Φ and

T'_Φ. If $rep(U) = \emptyset$, there does not exist valuation h of U_s that satisfies U_Φ. This is to say there does not exist a substition that is a valuation of T_s and satisfies T_Φ, and in the mean while, a valuation of T'_s and satisfies T'_Φ. In other words, $rep(T) \cap rep(T') = \emptyset$.

Suppose $S \in rep(U)$, then there is a valuation h that satisfies U_s. Therefore, h is a valuation of T_s that satisfies T_Φ. That is, $S \in rep(T)$. Similarly we have $S \in rep(T')$. Hence, $S \in rep(T) \cap rep(T')$.

Next, assume schema $S \in rep(T) \cap rep(T')$, we will show $S \in rep(U)$. Schema S is a valuation $h(T_s)$ that satisfies T_Φ. In the mean while, schema S is also a valuation $h(T'_s)$ that satisfies T'_Φ. So S is a valuation $h(U_s)$ that satisfies U_Φ. We showed $S \in rep(U)$.

Proved.

Definition Operator Merge

Operator Merge, denoted $\odot$, takes as input two schema tableaux T_1 and T_2, and generates an schema tableau $T_1 \odot T_2$, such that

$$rep(T_1 \odot T_2) = rep(T_1) \cap rep(T_2).$$

3.3 Rules to Compute Schema Tableaux

Let q be an SQL query generated by the grammar above, and let Θ_q be the parse tree of q. We shall associate a tableau (T, α) with each node of Θ_q.

Let ℓ be a leaf of Θ_q. We associate tableaux $(T, \alpha)(\ell)$ with ℓ as follows:

- if ℓ is of the form of attribute `A`, then $T(\ell) = \emptyset$, $\alpha(\ell) = \langle A\colon x \rangle$, and $T_\Phi(\ell) =$

$\{(A : \_) : \perp\}$.

- if ℓ is of the form of relation **r**, then $T(\ell) = \{r\{\}*\}$, $T_\Phi(\ell) = \emptyset$, and $\alpha(\ell) = r\{\}*$.

- For all other leafs ℓ, we set $T(\ell) = T_\Phi(\ell) = \emptyset$.

For the internal nodes of ν of Θ_Q we associate tableaux as follows:

1. Let ν correspond to rule

   `<SFW> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>` and let $\ell_1, \ldots, \ell_6$ be the children of ν, from left to right. Let U be the tableau with $U_s = \emptyset$, $U_\phi = \{(A : \_) : R_1 \vee \ldots \vee R_n : A \in attrs(\ell_2) \cup attrs(\ell_6)\}$. Then $T(\nu) = T(\ell_2) \odot T(\ell_4) \odot T(\ell_6) \odot U$

2. Let ν correspond to rule `<Query> ::= <attribute>,<SelList>` and let ℓ_1, ℓ_2 and ℓ_3 be the children of ν. Then $T(\nu) = T(\ell_1) \odot T(\ell_3)$.

3. Let ν correspond to rule `<SelList> ::= <attribute>` and let ℓ be the child of ν Then $T(\nu) = T(\ell)$.

4. Let ν correspond to rule `<attribute> ::= A` and let ℓ be the child of ν Then $T(\nu) = T(\ell)$.

5. Let ν correspond to rule `<FromList> ::= <Relation>` and let ℓ be the child of ν Then $T(\nu) = T(\ell)$.

6. Let ν correspond to rule `<FromList> ::= <Relation>, <FromList>` and let ν_1, ν_2 and ν_3 be the children of ν Then $T(\nu) = T(\nu_1) \odot T(\nu_2) \odot T(\nu_3)$.

7. Let ν correspond to rule `<Relation> ::= R` and let ℓ be the child of ν Then $T(\nu) = T(\ell)$.

8. Let ν correspond to rule `<Condition> ::= <attribute> = <attribute>` and let ν_1, ν_2, and ν_3 be the children of ν, from left to right. Let A_1 be the child of ν, and A_2 be the child of ν_3.

   Let $T(\nu') = \{(A_1 : x) : \perp, (A_2 : x) : \perp\}$. $T(\nu) = T(\ell_1) \odot T(\ell_2) \odot T(\ell_3)$.

Theorem 3 *Let Θ_q be the parse tree of query $q \in L(G_{SQL})$, and ν the root of $T(\Theta_q)$. Let $T(\nu)$ be the schema tableau associated with ν. Then*

$$rep(T(\nu)) = schemas(\nu).$$

Proof

Let $\nu_1, \ldots, \nu_6$ be the children of ν.

First we prove the soundness. Assume $\sigma \in rep(T(q))$. Let ν be the root of Θ_q. Then from the construction of $T(q)$, we know

$$T(\nu) = T(\nu_1) \odot \ldots \odot T(\nu_6) \odot T(\nu')$$

where

$$T(\nu') = \{(A : \_) : R_1 \vee \ldots \vee R_n \mid A \in Attrs(SelList) \cup Attrs(Condtion)\}.$$

Since $\sigma \in rep(T(q))$, by the definition of operator $\odot$, we have

$$\sigma \models T(\nu_1), \ldots, T(\nu_6), T(\nu').$$

By $\sigma \models T(\nu')$, $A \in attrs(\nu_2)$, there is an $R \in rels(\nu_4)$, and an $i > 0$, such that $\sigma(R, A, i)$ is defined. Furthermore, for all $A \in attrs(\nu_6)$, there is an $R \in rels(\nu_4)$, and an $i > 0$, such that $\sigma(R, A, i)$ is defined. And this exactly matches the definition of $\sigma \models \nu$, where ν is labelled `<SFW>`.

Then we prove the completeness. Assume $\sigma \models \Theta_q$. We need to show there exist an valuation ϑ such that $\sigma = \theta(T(q))$. Obviously, we have $\sigma \models \nu_1, \ldots, \nu_6$, and for all $A \in attrs(\nu_2)$, there is an $R \in rels(\nu_4)$, and an $i > 0$, such that $\sigma(R, A, i)$ is defined. Furthermore, for all $A \in attrs(\nu_6)$, there is an $R \in rels(\nu_4)$, and an $i > 0$, such that $\sigma(R, A, i)$ is defined. On the other hand, from the construction of $T(q)$, we know

$$T(\nu) = T(\nu_1) \odot \ldots \odot T(\nu_6) \odot T(\nu')$$

where

$$T(\nu') = \{(A : \_) : R_1 \vee \ldots \vee R_n \mid A \in Attrs(SelList) \cup Attrs(Condtion)\}.$$

Hence, there exist an valuation h of $T(v)$ that satisfies T_Φ. We showed $\sigma \in rep(T(q))$. Proved.

3.4 Implementation of Operator Merge

Definition Equivalent schema tableaux

Schema tableaux α and β are equivalent if and only if for every schema σ such that

$$\sigma \models \alpha$$

it is also true that

$$\sigma \models \beta$$

and vice versa.

Property of Merge

Operator Merge $\odot$ is idempotent, commutative and associative.

Procedure of Merging schema tableaux

But first, we shall define a few preliminary terms.

Definition Proper substitution

Let schema atom α is of one of the following forms

- $R\{X_1 : x_1, \ldots, X_n; x_n\}$;

- $R\langle X_1 : x_1, \ldots, X_n\rangle$;

- $R\{X_1 : x_1, \ldots, X_n : x_n\}*$.

Let $vdom(\theta)$ be the domain of variables in substitution θ, and $ran(\theta)$ the range. A substitution θ is said to be *proper for* α if and only if $vdom(\theta) \cap ran(\theta) = \emptyset$ and there exists a schema σ such that $\sigma \models \theta(\alpha)$.

The condition $vdom(\alpha) \cap ran(\theta) = \emptyset$ forces no inter-substitution among existing variables.

Example: If existing variables X and Y can replace one another, we introduce a new variable Z, for instance, and have substition $\{X/Z, Y/Z\}$.

Lemma 2 *Let α be an atom, and θ a proper substitution. Then $\theta(X) \neq \theta(Y)$ whenever $X \neq Y$.*

Let T be the schema tableau $\{\alpha_1, \ldots, \alpha_n\}$, for some $n \geq 0$. Then a substitution θ is said to be *proper substitution for* T if and only if there exists a schema σ such that $\sigma \models \theta(T)$.

Definition Isomorphism

Two schema atoms α and β are *isomorphic* if there exists a one-one onto substitution θ that maps variables to variables such that $\theta(\alpha) = \beta$.

In other words, α and β are the same up to renaming of variables.

Similarly, two schema tableaux T and T' are said to be *isomorphic* if there exists a one-one onto substitution θ that maps variables to variables such that $\theta(T) = T'$.

In other words, T and T' are the same up to renaming of variables.

Definition Most General Merger

A proper substitution θ is a *most general merger*, or *mgm*, of two schema atoms α and β, if and only if it can unify a maximal subset of the variables of α and β in a unique way, such that

$$rep(\alpha) \cap rep(\beta) = rep(\theta(\alpha)) \cap rep(\theta(\beta)).$$

It follows that θ is unique up to isomorphism.

Example: Assume schema atoms $\alpha = R\langle X_1 : x_1, X_2 : x_2, X_3 : x_1\rangle$, and $\beta = R\{A : int, B : int, Y : char\}$.

Substitutions $\theta = \{X_1/A, X_2/Y, X_3/C, x_1/int, x_2/char, x_3/int\}$ and $\theta' = \{X_1/C,$

$X_2/Y, X_3/A, x_1/int, x_2/char, x_3/int\}$ are both proper substitution of α and β. However, neither of θ or θ' is unique, hence neither is the mgm.

Intuitively, mgm of two schema atoms is the upper bound of the "sure" things of merging two schema atoms. In the previous example, $R\langle X_2/Y, x_2/char\rangle$ is a mgu of α and β.

We shall define mgm of schema tableaux similarly.

Definition Unifier of schema atoms

A proper substitution θ is a unifier of schema atoms α and β if $\theta(\alpha) = \theta(\beta)$.

A unifier is a special case of a mgm, which fully merges two atoms and results in two identical copies of atoms. On the other hand, a mgm may just partially merge two atoms, with some parts unresolved yet. If this is the case, we need to keep the unresolved parts, for example, in a condition box.

The set of AD pairs in schema atoms α and β that remain not unified after being applied a mgm θ is denoted $AD_{unsure}(\alpha, \theta)$, and $AD_{unsure}(\beta, \theta)$.

In the previous example, $AD_{unsure}(\alpha, \theta) = \{(A : int), (C : int)\}$.

A subroutine: Finding mgm of two schema atoms and a condition box

This is a major subroutine we shall use in operator Merge. The goal is to merge α and β as much as possible without losing the generality, such that the condition box is satisfied. That is to find out the "sure" things about the structure of two atoms have in common.

Let T_Φ be a condition box. Let α and β be two schema atoms, each in one of the following forms:

1. $R\langle X_1 : x_1, \ldots, X_n : x_n\rangle$;

2. $R\{X_1 : x_1, \ldots, X_n : x_n\}$;

3. $R\{X_1 : x_1, \ldots, X_n : x_n\}*$.

The steps of finding mgm is as follows:

1. Choose one atom whose number of AD pair is not fewer than the other. Let assume it is α.

2. For every AD pair in β, unify one distinct AD pair in α with some substitution θ_i, such that no attribute occurs twice in the resulting atom. Discard the resulting atom if it does not satisfy T_Φ. Let θ be the composite substitution for a survived resulting atom. We have $\theta = \bigcup \theta_i$.

3. Look up the results in previous step, find out those results that hold for all cases. For each of such results, find out the necessary substitution that contributes, say θ_j.

4. Composite substitution $\theta = \bigcup \theta_j$ is a mgm.

This is a naive algorithm that remains to be optimised.

Procedures of Merge Operator

Now we are ready to define the procedures of operator Merge.

Given schema tableaux T_1, and T_2. Then $T_1 \odot T_2$ is defined as follows:

1. The result of merging two schema tableaux is a schema tableau, say T, preliminarily initialised to $\emptyset$.

2. Resolve variables in T_1 and T_2, such that the set of variables used in T_1 and that in T_2 are disjoint.

3. Add every schema atom from T_1 and T_2 to T.

4. Add condition boxes $T_\Phi(T_1)$ and $T_\Phi(T_2)$ to condition box $T_\Phi(\nu)$.

5. Repeat {

   (a) Let $T_{old} = T$.

   (b) For every pair of schema atoms α and β in T, such that $r(\alpha) = r(\beta)$, find mgm θ of α and β.

   Apply θ to T, i.e. $\theta(T)$.

   (c) If one atom in previous step is in the form $R_i\langle ., \ldots, .\rangle$, say α, keep $\theta(\alpha)$ in T_s; for every AD pair $(X : x)$ that are not fully merged in the other participating atom, add $(\theta(X) : \theta(x)) : R_i$ to T_Φ.

   (d) If none of the participating atoms is in the form $R_i\langle ., \ldots, .\rangle$, find one atom whose number of AD pair is not fewer than the other, say α. Keep $\theta(\alpha)$ in T_s; for every AD pair $(X : x)$ that are not fully merged in the other participating atom, add $(\theta(X) : \theta(x)) : R_i$ to T_Φ.

   (e) In condition box T_Φ, if there are more than one identical AD pair $(X : x)$ with different boolean expression ϕ_i, replace them by $(X : x) : \bigwedge \phi_i$. If

```
        there are AD pairs $(X : x_i) : R$ and $(X : x_j) : R$, for $i \neq j$, then we have $\theta$
        to unify $x_i$ and $x_j$. Apply $\theta$ to $T_s$ and $T_\Phi$.
    }while $(T \neq T_{old})$
```

3.5 Extension of Grammar G_{SQL}

Having built a well-defined framework for type inference on a small subset of the standard SQL, we would extend it to include more SQL features. Since the standard SQL is a very big family, currently we would consider the subset that is sufficient to solve textbook examples. Some of them may be readily to be added to the existing G_{SQL}, such as `<Query> ::= <SFW> Except <SFW>`. Others may need some extra care. Correlated nested queries, for example, seems to add certain degree of complexity to the problem. However, all nested queries can be flattened and remain essentially the same. A detailed treatment can be found in the report [7].

Chapter 4

Applications and Experiments

This chapter demonstrates applications and experiments of our work. We design two artificial databases in each of which there exists a problem that will be tackled by our work. In the first case, we introduce the architecture of a system to recover database, as much as possible, from a set of queries. Consisting of a parser, a type inference engine, and a conjugator, the system takes as input a collection of SQL queries and generates a schema tableau of the database. In the second case, we employ the schema tableau to build a data structure to help manage stored procedures. Solving these problems is motivation of our work, too.

We also perform the initial set of experiments on realistic inputs, the suite of queries of TPC-H, which has broad industry-wide relevance with a high degree of complexity. The experiment results are analysed and summarised as well.

4.1 Application: Discovering the Database

Assume we have a crucial application which relies on some antiquated RDBMS. For some reasons, say, the vendor ran out of business, and the system is no more supported. Or even worse, it crashed before the user has the chance to migrate to a modern DBMS. Since the application is critical , the user cannot afford to lose the data and have to recover the system as much as possible. Fortunately, there is a reasonably comprehensive set of queries available, for example, embedded in some host language. In such a situation, our work could come to the rescue. Incorporated with other techniques of reverse engineering, we believe the system can be restored to some extent.

The Architecture

The steps of the discovering are illustrated in Figure 4. Each query is fed as input to the SQL parser, which will generate a regular parse tree accordingly. The parse tree is then run on the type inference engine and results in tableau, which specifies the database schema information inferable from the SQL expression. The tableaux associated with roots of parse trees are merged to form a tableau for the set of queries under discussion. Usually, there are many instantiations that satisfy the tableau.

Outputs and Analysis

Attributes, data types, and positions can be reconstructed to some extend by this way, depending on the 'quality' of the queries. The more comprehensive the queries are,

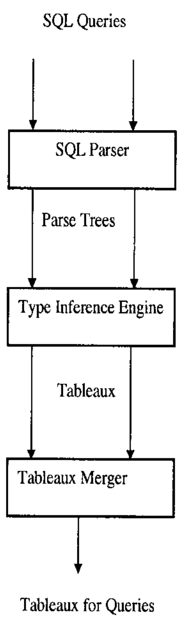


Figure 4: From Queries to Tableaux

the more we will find out. Ideally, we hope the scopes of `<FromList>` as restricted as possible, namely the relations involved as few as possible, and the number of attributes as many as possible. This is the most direct way to locate the scopes of attributes. If an attribute is known to be in two scopes, independently inferred from different sources, then we could further narrow down the scope of the attribute to the overlapping part of the two, as the merger does. This is based on the assumption that attribute names are unique across the database.

Additionally, we identify a set of candidate pairs for relationships among tables. We argue that the attributes that are compared in where clause are good candidates for relationships, in that we often take use of relationships by comparing attributes on two sides of relationships. However, this is just one side of the situation. While we tend to compare two sides of relationships, all comparisons are not necessarily involved in attributes of relationships. Nonetheless, an entity in third normal form would represent only one theme, it is less likely there are more than one attribute with same semantics.

Assume we have tables *students* and *marks*:

`students(Fname char(10), Lname char(10), ID char(10), Phone char(10))`,

and

`marks(StudentID char(10), Course char(10), Mark double).`

While nothing prevents one posing an ad hoc query such as

`Select Phone From students, marks Where Fname = Course.`

since attributes Fname and Course are of same type, we expect more meaningful queries such as

`Select ID, Course, Mark From students, marks Where ID = StudentID.`

As for the multiplicity of relationships, it cannot be decided by this work itself. To ultimately determine it, we need to look into other available information. For example, investigation at instance level can provide information that cannot be obtained from schema level. Linguistic analysis technique is used to evaluating description in some prototypes, too. In fact, this is a schema matching issue that attracts more and more attention [13].

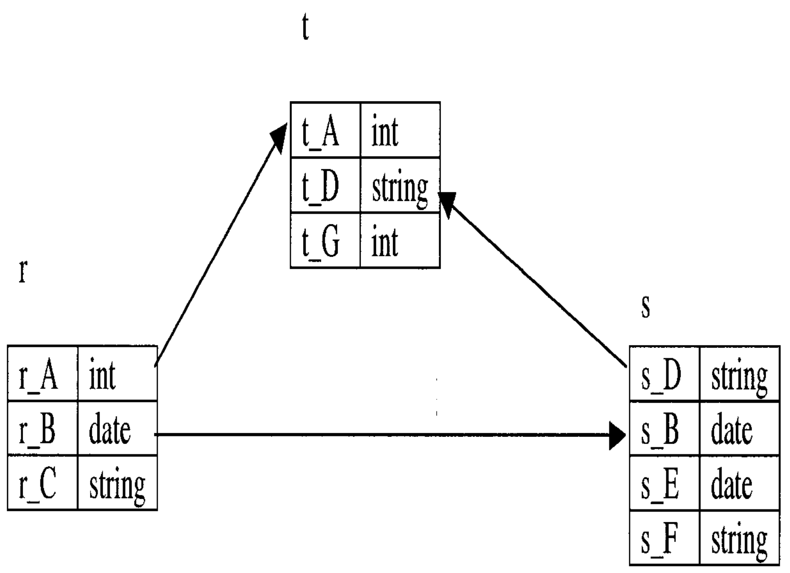


Figure 5: An Illustrative Example

An Illustrative Example

Fix a database schema as shown in Figure 5, so that we can compare it with what we will discover. The arrows point in the direction of one-to-many relationships between tables. There are three tables, R, S and T. The naming convention is attributes are preceded by table names, such that each attribute name is unique across the database. We treat each attribute name as a string and do not infer any information from the table names. For example, we do not know attribute R_A is from table R by the string itself.

We assume the DBMS supports only the following types: *int*, *double*, *date*, *string*. The types *int* and *double* are comparable as assumed elsewhere. Hence they are subtypes of a more general type, called *number*.

The set of queries and their type formulas are:

1. Query 1

```
Select R_A, R_B

From R, T

Where  R_A = T_A

               and T_D like '%computer%'
```

Its schema tableau is shown in Figure 6.

		$(R\_A : x) : R \vee T$ $(T\_A : x) : R \vee T$ $(R\_B : \_) : R \vee T$ $(T\_D : string) : R \vee T$
R	{}*	
T	{}*	

Figure 6: Schema Tableau of Query 1

2. Query 2

```
Select T_D, S_B

From T, S

Where T_D = S_D

       And S_B > (

           Select R_B

           From R

           Where R_A = 100

           )
```

Its schema tableau is shown in Figure 7.

3. Query 3

		$(T\_D : x) : T \vee S$
		$(S\_B : y) : T \vee S$
		$(S\_D : x) : T \vee S$
R	$\{R\_A : number, R\_B : y\}*$	
S	$\{\}*$	

Figure 7: Schema Tableau of Query 2

```
Select S_D

From R, S

Where  R_B Between S_B And S_E

        And S_E < date '1998-10-22'
```

Its schema tableau is shown in Figure 8.

		$(S\_D : x) : R \vee S$
		$(R\_B : date) : R \vee S$
		$(S\_B : date) : T \vee S$
		$(S\_E : date) : T \vee S$
R	$\{\}*$	
S	$\{\}*$	
T	$\{\}*$	

Figure 8: Schema Tableau of Query 3

		$(T\_A : number) : R \vee T$
		$(S\_E : date) : R \vee S$
R	$\{R\_A : number, R\_B : date\}*$	
T	$\{T\_D : string\}*$	
S	$\{S\_D : x, S\_B : date\}*$	

Figure 9: Schema tableau of Queries 1, 2, and 3

Schema tableau obtained from Queries 1, 2 and 3 is shown in Figure 9.

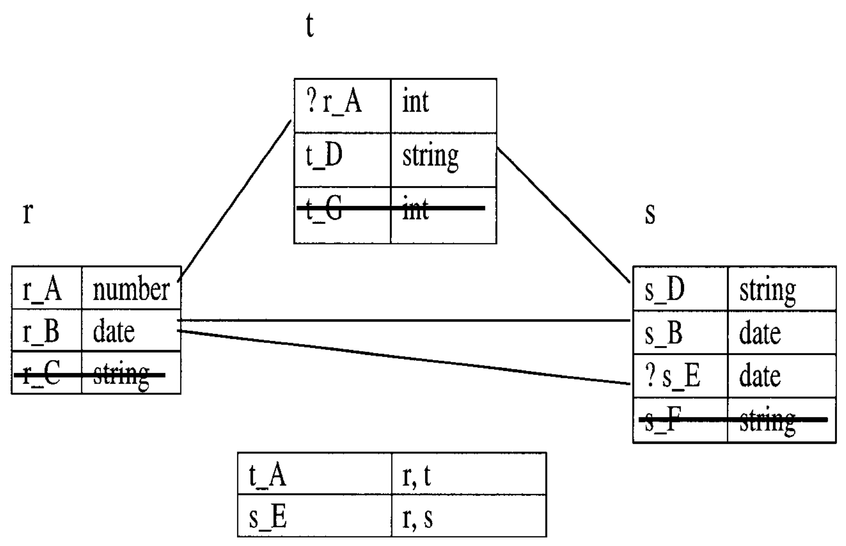


Figure 10: Database Schema Comprehension – Example

The result is shown in Figure 10. If an attributes is preceded by a question mark, it means the known scope of the attribute cover more than one tables. The attribute belongs to only one tables, but we do not know which one precisely. *R_A* and *S_E* are such attributes. *R_A* is possible from r or t, while *S_E* is possible from *R* or *S*. There are also attributes with a bar, such as *R_C*, which means we do not know the existence of them.

4.2 Application: Stored Procedure Maintenance

A stored procedure is a precompiled collection of Transact-SQL statements stored under a name and processed as a unit that users can call from within another Transact-SQL statement or from the client applications .

Using stored procedures has several advantages over giving direct users direct access to the underlying data [16]. Firstly, it improves the performance. Stored Procedures run quickly because they do not need to repeat parsing, optimising and compiling with each execution. Moreover, stored procedures reduce the loading of client computer and the network traffic, which results in increasing speed, as they run on the server side. Secondly, Stored procedures can be used to enhance security and hide underlying data objects from unauthorised users. For example, you can give the users permission to execute the stored procedure to work with a restricted set of the columns and data, while not allowing permissions to select or update underlying data objects. By using the store procedures, the permission management could also be simplified. You can grant `EXECUTE` permission on the stored procedure instead of granting permissions on the underlying data objects. Thirdly, Stored procedures can be used to enhance the reliability of your application. For example, if all clients use the same stored procedures to update the database, the code base is smaller and easier to troubleshoot for any problems. In this case, everyone is updating tables in the same order and there will be less risk of deadlocks. Stored procedures can be used to conceal the changes in database design too. For example, if you denormalize your database design to provide faster query performance, you need only to change the stored procedure, but applications that use the results returned by this stored procedure, will not be rewritten.

Assume we have tables R and S, with the following schemas

$$R : \langle (A, int), (B, char(20)), (C, int) \rangle$$

and

$$S : \langle (D, int), (E, int), (F, char(20)) \rangle .$$

And we have the query:

```
Select A, D
From R, S
Where A = F
      And A = 10
      And E like '%tom%'
```

where 10 and *tom* are user inputs.

Assume data types the DBMS supports are int, double, char. And int is comparable with double. Let us call the family of int and double number. Using our algorithm, the type formula of this query is shown in the following tableau.

		$(A : number) : R \vee S$ $(D :) : R \vee S$ $(E : char) : R \vee S$ $(F : number) : R \vee S$
R	{}*	
S	{}*	

Now we write an equivalent stored procedure, that takes two parameters. Syntax of creating stored procedures differs from one DBMS to another, but most would be similar to the following:

```
Create or Replace Procedure myProc
     input1 IN int,
     input2 IN char(10),
     output1 OUT int,
```

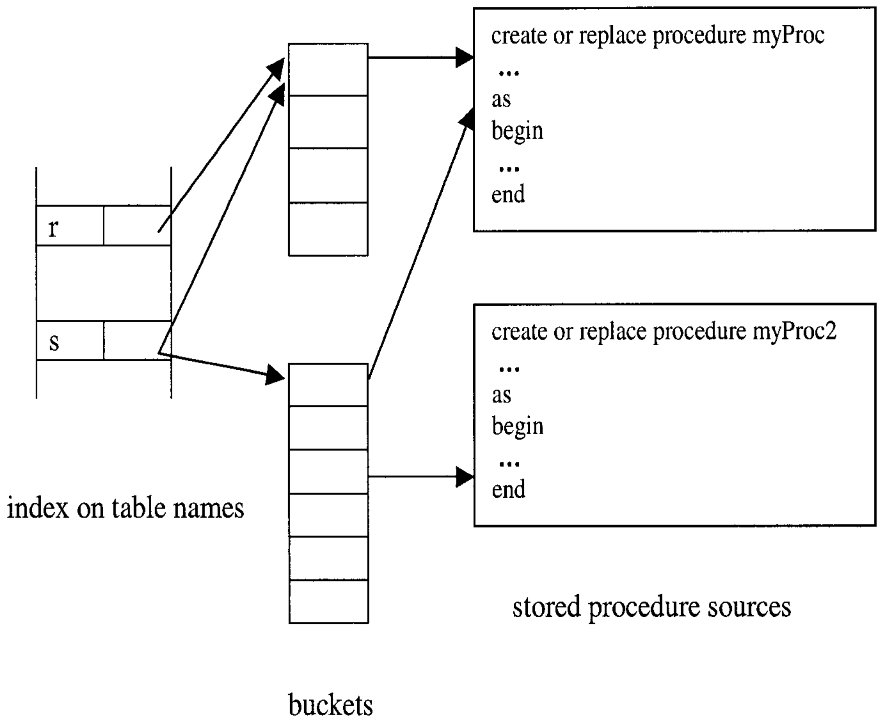


Figure 11: Maintenance of Stored Procedures

```
     output2 OUT char(20)
As
Begin
Select A, D
     into output1, output2
From R, S
Where A = F
     And  A > input1
     And  E like (%input2%)
End
```

If the structures of R and S are changed, this query may or may not be valid. If column C is dropped, it is not affected. However, if column E is dropped, then the query is not valid any more. This is also the case of the stored procedure.

When the schema of a table changes, the DBMS should respond accordingly to maintain the integrity of the database. Among what needed to be examined are those stored procedures related to the table and ensure these stored procedures still valid, and prompt the user to take proper step when the schema change affects the validity of some stored procedures. The type formulas of stored procedures can be used as a tool to help perform this task efficiently, along with other techniques such as indexing on tables. Figure 11 is an illustration of the idea.

4.3 Experiments on TPC-H

This section describes the initial set of experiments concerning the usability of our algorithm. We design a system that takes as input a collection of SQL expression and generates a type context of the database. The system consists of a parser, and type inference engine, and a conjugator. Then we run the system manually on the suite of queries of TPC-H, which has broad industry-wide relevance with a high degree of complexity. The results are presented.

What is TPC-H

The TPC Benchmark^{TM} H (TPC-H) is a decision support benchmark. It consists of a suite of business oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance [18].

This benchmark illustrates decision support systems that

- Examine large volumes of data;
- Execute queries with a high degree of complexity;
- Give answers to critical business questions.

The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size), and reflects multiple aspects of the capability of the system to process queries. These aspects include the selected database size against which the queries are executed, the query processing power when queries are submitted by a single stream, and the query throughput when queries are submitted by multiple concurrent users. The TPC-H Price/Performance metric is expressed as \$/QphH@Size.

Why Use TPC-H Queries As Input?

We choose TPC-H queries as input for our algorithm because they are representative of complex business analysis applications. They have been given in a realistic context, portraying the activity of a wholesale supplier to help the reader relate intuitively to the components of the benchmark. The queries have following characteristics:

- Give answers to real-world business questions;
- They are of an ad hoc nature;
- Are far more complex than most OLAP transactions;

- Include a rich breadth of operators and selectivity constraints;
- Generate intensive activity on the part of the database server component of the system under test;
- Are executed against a database complying to specific population and scaling requirements;
- Are implemented with constraints derived from staying closely synchronised with an on-line production database.
- They all differ from each other;

Business and Application Environment

TPC-H does not represent the activity of any particular business segment, but rather any industry which must manage , sell or distribute a product worldwide (e.g., car rental, food distribution , parts, suppliers, etc.). TPC-H does not attempt to be a model of how to build an actual information analysis application.

These selected queries provide answers to the following classes of business analysis:

- Pricing and promotions;
- Supply and demand management;
- Profit and revenue management;
- Customer study;

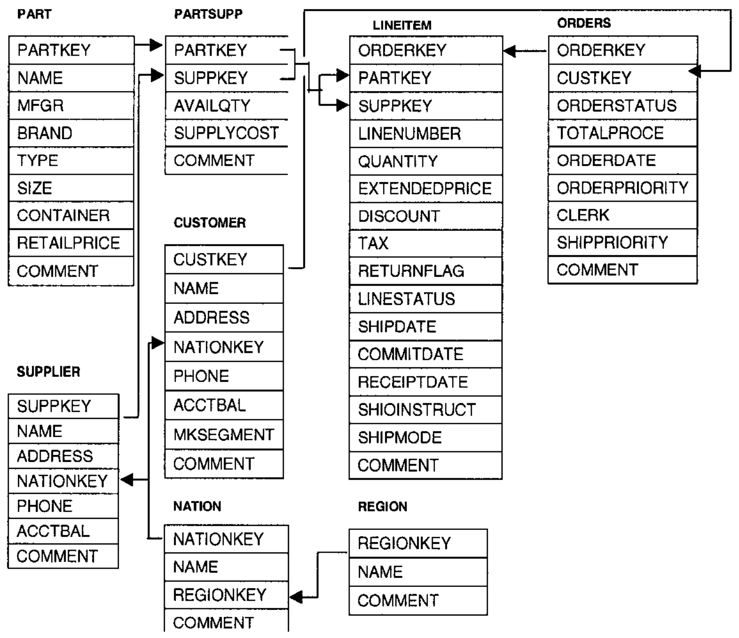


Figure 12: TPC-H Schema

- Market share study;
- Shipping management;

Database Entities, Relationships, and Characteristics

The components of the TPC-H database are defined to consist of eight separate and individual tables (the Base Tables). The relationships between columns of these tables are illustrated in Figure 12.

A Glimpse of A Few Queries

We provide details of a few queries and the primitive output of each query, followed by a summary. The reader may want to find out details of other queries in the Appendix.

Note the output of a query is a view and it is less relevant in terms of discovering attribute information from base tables, unless it is fed as an input of next operation. We show the more relevant type context only for each query, for brevity too.

In TPC-H, the attribute names are proceeded by relation names, for example, n_nationkey and r_nationkey, such that each attribute is unique across the database. We adopt this assumption as well.

1. Pricing Summary Report Query (Q1)

   This query reports the amount of business that was billed, shipped, and returned.

   (a) Business Question

      The pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60-120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by `returnflag` and `linestatus`, and listed in ascending order of `returnflag` and `linestatus`. A count of the number of lineitems in each group is included.

   (b) Functional Query Definition

```
Select
   l_returnflag,
   l_linestatus,
   Sum(l_quantity) As sum_qty,
      Sum(l_extendedprice) As sum_base_price,
      Sum(l_extendedprice*(1-l_discount)) As sum_disc_price,
      Sum(l_extendedprice*(1-l_discount)*(1+l_tax)) As sum_charge,
      Avg(l_quantity) As avg_qty,
      Avg(l_extendedprice) As avg_price,
      Avg(l_discount) As avg_disc,
      Count(*) As count_order
   From
      lineitem
   Where
      l_shipdate <= date '1998-12-01' - interval 90 day
   Group By
      l_returnflag,
      l_linestatus
   Order By
      l_returnflag,
      l_linestatus;
```

(c) The schema tableau is shown in Figure 13.

L	$\{l\_returnflag : \_, l\_linestatus : \_, l\_quantity : \_, l\_tax : \_$ $l\_extendedprice : \_, l\_discount : \_, l\_shipdate : date\}*$	

Figure 13: Schema Tableau of Query 1

2. Minimum Cost Supplier Query (Q2)

This query finds which supplier should be selected to place an order for a given part on a given part in a given region.

(a) Business Question

The Minimum Cost Supplier Query finds, in a given region, for each part of a certain type and size, the supplier who can supply it at minimum cost. If several suppliers in that region offer the desired part type and size at the same (minimum) cost, the query lists the parts from supplies with the 100 highest account balances. For each supplier, the query lists account balance of the supplier, name and nation; the part's number and manufacturer; the supplier's address, phone number and comment information.

(b) Functional Query Definition

```
Select
    s_acctbal,
    s_name,
    n_name,
    p_partkey,
    p_mfgr,
    s_address,
    s_phone,
    s_comment
From
    part,
    supplier,
    partsupp,
    nation,
    region
Where
    p_partkey = ps_partkey
    And s_suppkey = ps_suppkey
    And p_size = 15
    And p_type like '%Brass'
    And s_nationkey = n_nationkey
    And n_regionkey = r_regionkey
    And r_name = 'Europe'
    And ps_supplycost = (
          Select
              Min(ps_supplycost)
          From
              partsupp, supplier,
              nation, region
          Where
              p_partkey = ps_partkey
              And s_suppkey = ps_suppkey
              And s_nationkey = n_nationkey
              And n_regionkey = r_regionkey
```

```
                  And r_name = 'Europe'
                  )
Order By
    s_acctbal desc,
    n_name,
    s_name,
    p_partkey;
```

(c) The schema tableau is shown in Figure 14.

		$(s\_acctbal : \_) : P \vee S \vee PS \vee N \vee R$
		$(s\_name : \_) : P \vee S \vee PS \vee N \vee R$
		$(n\_name : \_) : P \vee S \vee PS \vee N \vee R$
		$(p\_partkey : x_1) : P \vee S \vee PS \vee N \vee R$
		$(p\_mfgr : \_) : P \vee S \vee PS \vee N \vee R$
		$(s\_address : \_) : P \vee S \vee PS \vee N \vee R$
		$(s\_phone : \_) : P \vee S \vee PS \vee N \vee R$
		$(s\_comment : \_) : P \vee S \vee PS \vee N \vee R$
		$(s\_supplykey : x_2) : P \vee S \vee PS \vee N \vee R$
		$(ps\_supplykey : x_2) : P \vee S \vee PS \vee N \vee R$
		$(ps\_partkey : x_1) : P \vee S \vee PS \vee N \vee R$
		$(p\_size : number) P \vee S \vee PS \vee N \vee R$
		$(p\_type : char) P \vee S \vee PS \vee N \vee R$
		$(s\_nationkey : x_3) P \vee S \vee PS \vee N \vee R$
		$(n\_nationkey : x_3) : P \vee S \vee PS \vee N \vee R$
		$(n\_regionkey : x_4) : P \vee S \vee PS \vee N \vee R$
		$(r\_regionkey : x_4) : P \vee S \vee PS \vee N \vee R$
		$(r\_name : char) : P \vee S \vee PS \vee N \vee R$
		$(ps\_supplycost : \_) : S \vee PS \vee N \vee R$
P	{}*	
R	{}*	
N	{}*	
S	{}*	
PS	{}*	

Figure 14: Schema Tableau of Query 2

3. Shipping Priority Query (Q3)

This query retrieves the 10 unshipped orders with the highest value.

(a) Business Question

The shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of `l_extendedprice * (1-l_discount)`, of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more that 10 unshipped orders exist, only the 10 orders with the largest revenue are listed.

(b) Functional Query Definition

```
Select
    l_orderkey,
    Sum(l_extendedprice*(1-l_discount)) As revenue,
    o_orderdate,
    o_shippriority
From
    customer,
    orders,
    lineitem
Where
    c_mktsegment = 'Building'
    And c_custkey = o_custkey
    And l_orderkey = o_orderkey
    And o_orderdate < date '1995-03-15'
    And l_shipdate > date '1995-05-15'
Group By
    l_orderkey,
    o_orderdate,
    o_shippriority
Order By
    revenue Desc,
    o_orderdate;
```

(c) Schema tableau is shown in Figure 15

		$(o\_orderdate : date) : C \vee O \vee L$
		$(o\_shippriority : \_) : C \vee O \vee L$
		$(c\_marketsegment : char) : C \vee O \vee L$
		$(l\_shipdate : date) : C \vee O \vee L$
		$(l\_extendedprice : \_) : C \vee O \vee L$
		$(l\_discount : \_) : C \vee O \vee L$
		$(l\_orderkey : x_2) : C \vee O \vee L$
		$(c\_custkey : x_1) : C \vee O \vee L$
		$(o\_custkey : x_1) : C \vee O \vee L$
		$(o\_ortherkey : x_2) : C \vee O \vee L$
L	{}*	
C	{}*	
O	{}*	

Figure 15: Schema Tableau of Query 3

Findings of the Experiment

As mentioned earlier, one goal of the experiment is to find out as much information of attributes and relations as possible. We want to know what attributes there are in a certain table. If we can locate it exactly, we want to know the scope of the attribute as precisely as possible. The test exhibits a very good results in terms of the amount of such information discovered. We found 35 out of 61 attributes are located exactly in the relation they belong to, which amounts to a ratio of 57.3 % of all the attributes. The scope of 8.2% of attributes is limited to two relation; namely, these attributes exist in one of two relations we know, but we don not know which relation they exactly exist. In total, the scope of 29.5% of attributes are narrowed down to some extent, in addition to the 57.3% that are precisely determined. We unsurprisingly have no clue about the rest of 13.1% of attributes, which are not mentioned at all.

Attribute Scope In	Number of Attributes	Percentage (%)
1 Relation	35	57.37
2 Relations	5	8.2
3 Relations	7	11.46
4 Relations	4	6.55
5 Relations	2	3.27
N.A	8	13.11

Table 1: Summary of All Relations

Note the assumption adopted is every attribute name is unique, which is a restrict one. Under such assumption, *n_nationkey* and *r_nationkey* are two distinct attributes, and we don't infer any extra information, for instance, from an expression `n_nationkey = r_nationkey` except their data type are the same. Realistic assumptions are usually more relaxed than this. It is a convention to proceed a certain subset of attributes by relation names, hence it implies attribute *nationkey* appears in both relation N and R by the expression `n.nationkey =r.nationkey`. Therefore, one can expect more information are inferable if a more realistic assumption is adopted.

The result is summarised in Table 1. The detail of each table is shown in Table 3 to Table 10 in the next subsection.

As for the data types of the attributes, we find out the data type family of 22 attributes, that amounts to 36.1% of all attributes. By data type family, we mean we do not know the precise data type. To pinpoint the exact data type by SQL expression, we need to know the built-in data types of the DBMS under examination, and we need some heuristics and estimation. For example, by `o_orderdate >= date '12-Mar-2001'`, we can conclude that attribute *o_orderdate* is of type *date*, if we know the DBMS does use it to describe the general concept of time. Similarly, by `n_name = 'Japan'`, we can say attribute *n_name* is of *char*. However, we should be careful to infer the data type of attribute *l_quantity* from expression `l_quantity = 3`. It could be *int* or *real*, if the DBMS supports both. Nonetheless, since *int* and *real* are in the same family of hierarchy, and thus comparable, in most DBMS. We are safe to say it is of this type family of hierarchy. If more information is available, such as `l_partkey = 'computer'`, then we may be able to arrive at a conclusion closer to the truth based on our experience.

Other facts gathered by our work lead to some interesting finding as well. Obtaining knowledge of the relationships of an information system allows a better understanding of it. Researchers proposed an array of diversified approaches to extract relationships through database reverse engineering [15]. Experimental results of our work indicates our finding provides important evidence in determine relationships.

Our Finding	Relationships
$\mathcal{T}(p\_partkey) = \mathcal{T}(ps\_partkey)$	$1:n$
$\mathcal{T}(s\_suppkey) = \mathcal{T}(ps\_suppkey)$	$1:n$
$\mathcal{T}(n\_nationkey) = \mathcal{T}(s\_nationkey)$	$1:n$
$\mathcal{T}(n\_nationkey) = \mathcal{T}(c\_nationkey)$	$1:n$
$\mathcal{T}(ps\_partkey) = \mathcal{T}(l\_partkey)$	$1:n$
$\mathcal{T}(ps\_suppkey) = \mathcal{T}(l\_partkey)$	$1:n$
$\mathcal{T}(c\_custkey) = \mathcal{T}(o\_custkey)$	$1:n$
$\mathcal{T}(o\_orderkey) = \mathcal{T}(l\_orderkey)$	$1:n$
$\mathcal{T}(l\_commitdate) = \mathcal{T}(l\_receiptdate$	None

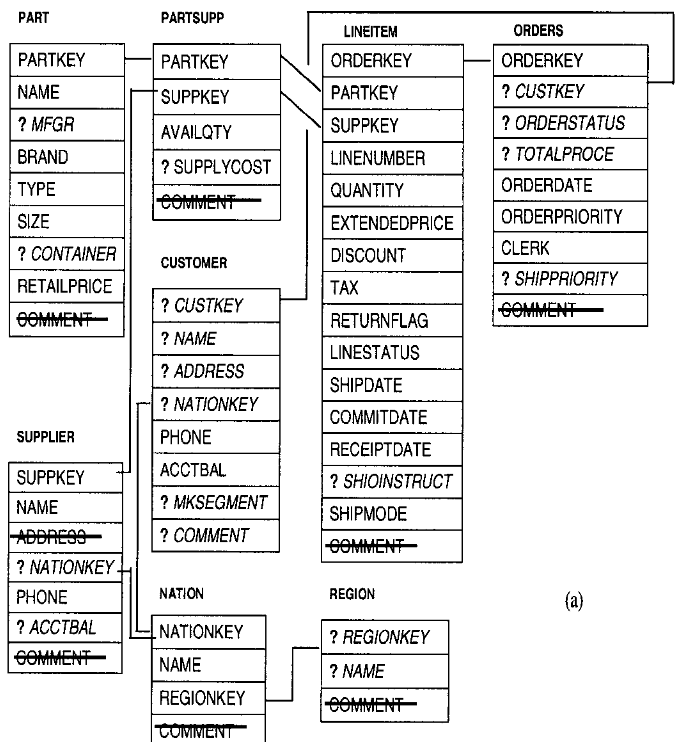
Table 2: Relationships Discovery

We note that the multiplicities are not determined by our work. Other techniques, such as exploring instance-level data, can give important insight into the contents and meaning of schema elements. This is especially true when useful schema information is limited [5].

However, we barely find any information concerning the position of the attributes and the arity of the relations. This is because the information of position and arity is inferred for the union operator, which does not exist in this set of queries. This finding indicates the position of attribute is an implementation issue, which does not have impact on all queries.

Figure 16 illustrates our discovery of the eight tables in a comparable manner to the actual database. In Figure 16-(a), some attributes are preceded by a question mark, which means there is evidence that those attributes may be part of the tables. Attributes *p_container* and *ps_supplycost* are such attributes. The degree of certainty of this belief is different from one to another of these attributes, which is summarised in Figure 16-(b). For example, *p_container* is possible from one of P or L, while the scope of *ps_supplycost* is one of P, S, or N. There are also attributes with a bar, such as *p_partsupp*, which means we do not know the existence of them.

Other results of the experiments are listed here, including the discovery of data type, and the scopes of attributes for each relation.



(a)

P_CONTAINER	? P, L
S_NATIONKEY	? S, N
L_LINENUMBER	? P, L
C_CUSTKEY	? C, O
O_CUSTKEY	? O, C
PS_SUPPLYCOST	? PS, S, N
C_NAME	? C, L, O
C_MKTSEGMENT	? C, L, O
O_TOTALPRICE	? O, C, L
O_SHIPPRIORITY	? O, C, L
R_REGIONKEY	? R, S, N
R_NAME	? R, S, N
C_ADDRESS	? C, L, O, N
C_NATIONKEY	? C, L, O, N
C_COMMENT	? C, L, O, N
O_ORDERSTATUS	? O, S, L, N
P_MFGR	? P, S, PS, N, R

(b)

Figure 16: TPC-H Database Schema Comprehension

Relation	Attribute	Discovery
Part	Partkey	P
	Name	P
	MFGR	P ∨ S ∨ PS ∨ N ∨ Re
	Brand	P
	Type	P
	Size	P
	Container	P ∨ L
	Retailprice	
	Comment	

Table 3: Summary of Relation Part

Relation	Attribute	Discovery
Supplier	Suppkey	S
	Name	S
	Address	
	Nationkey	S ∨ N
	Phone	S
	Acctbal	P∨ S∨PS∨ N∨ R
	Comment	S

Table 4: Summary of Relation Supplier

Relation	Attribute	Discovery
Partsupp	Partkey	PS
	Suppkey	PS
	Availqty	PS
	Supplycost	S ∨ PS ∨ N
	Comment	

Table 5: Summary of Relation Partsupp

Relation	Attribute	Discovery
Lineitem	Orderkey	L
	Partkey	L
	Suppkey	L
	Linenumber	
	Quantity	L
	Extendedprice	L
	Discount	L
	Tax	L
	Returnflag	L
	Linestatus	L
	Shipdate	L
	Commitdate	L
	Receiptdate	L
	Shipinstruct	$P \vee L$
	Shipmode	L
	Comment	

Table 6: Summary of Relation Lineitem

Relation	Attribute	Discovery
Costumer	Custkey	$C \vee O$
	Name	$L \vee C \vee O$
	Address	$L \vee C \vee O \vee N$
	Nationkey	$L \vee C \vee O \vee N$
	Phone	C
	Acctbal	C
	Mktsegment	$L \vee C \vee O$
	Comment	$L \vee C \vee O \vee N$

Table 7: Summary of Relation Costumer

Relation	Attribute	Discovery
Orders	Orderkey	O
	Custkey	$O \vee C$
	Orderstatus	$S \vee L \vee O \vee N$
	Totalprice	$C\vee O\vee L$
	Orderdate	O
	Orderpriority	O
	clerk	C
	shippriority	$C\vee O\vee L$
	Comment	

Table 8: Summary of Relation Orders

Relation	Attribute	Discovery
Nation	Nationkey	N
	Name	N
	Regionkey	N
	Comment	

Table 9: Summary of Relation Nation

Relation	Attribute	Discovery
Region	Regionkey	$S \vee N \vee R$
	Name	$S \vee N \vee R$
	Comment	

Table 10: Summary of Relation Region

Chapter 5

Conclusions and Future Work

We have studied the SQL version of type inference problem: given an SQL query q, under which database schema q is well-typed? We have developed a framework for type inference in SQL. First we have formally defined schema as a mapping from *Rels*, *Attrs* and $\mathbb{N}$ to a data type τ. Then we have considered grammar G_{SQL} that covers the core features of the standard SQL. The important characteristic is that there is a set of semantic rules besides the syntactic rules such that a "meaning" is assigned to string generated by the grammar. Furthermore, we have shown G_{SQL} is unambiguous and well-defined. We have defined when a schema S satisfies a query string q.

Since potentially we need to deal with incomplete database schemas, we have proposed a powerful conditional-table-like representation called schema tableau, which are associated with nodes of the derivation trees of query strings. We have also introduced an important operator, merge, on schema tableaux, and a set of rules to compute schema tableaux.

We have established that schemas satisfying nodes of the derivation trees can be represented by schema tableau. Furthermore, the schema tableau can be compute using the rules devised.

In addition to the theoretical significance, we have shown our work fits in many applications from reverse engineering to recent trend of programming such as maintenance of stored procedures. Having illustrated the idea in a few artificial database applications, including a manner to merge schema tableaux obtained from each query string, we have applied the approach on the suite of TPC-H queries which has broad industry-wide relevance with a high degree of complexity. The experiment and results indicate the framework proposed is useful for database schema comprehension. In this experiment, the scopes of 57% of attributes can be precisely located, in addition to 29.5% of attributes whose scopes can be narrowed down to some extent. This system also generates quality candidates of relationships among tables. The results exhibits an potential that in a broader context such as automatic schema matching, the framework proposed can land itself as a tool as well.

Expanding the set of inference rules to cover other SQL statements is among topics of future work. In this work, we have focused on a typical subset of SQL query language. But the concept can migrate smoothly to this goal. We would like to implement the system described in Figure 4 such that it can run on real DBMS. We would like to explore the complexity of testing whether a given query q is well-typed. The typability of a query can be efficiently reduced to satisfiability of MFO sentences. And the latter problem is known to be decidable in non-deterministic time $2^{O(n/logn)}$. It remains to be investigated whether we can improve it. Type checking attracts attention in semistructured data lately [17], and we would like to look into related issues in the context of semi-structured data models rather than the relational data model can be a avenue of future work. Query languages for these models are essentially schema-independent, since the assumption of a given fixed schema is relaxed or even abandoned. Nevertheless, querying is more effective if at least some form of schema is available, computed from the particular instance [4, 2].

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] Peter Buneman, Susan B. Davidson, Mary F. Fernandez, and Dan Suciu. Adding structure to unstructured data. In *Proceedings of the 6th International Conference on Database Theory*, pages 336–350. Springer-Verlag, 1997.

[3] Peter Buneman and Atsushi Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, 1996.

[4] Jan Van den Bussche and Emmanuel Waller. Type inference in the polymorphic relational algebra. In *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania*, pages 80–90. ACM Press, 1999.

[5] H. Do and E. Rahm. Coma - a system for flexible combination of schema matching approaches, 2002.

[6] Grahne G. *Problem of Incomplete Information in Relational Databases.* Springer-Verlag New York, Inc., 1991.

[7] Grahne G., Shiri N. and Lin W. Schemas and schema tableaux. Tech. report, Concordia University, Montreal, Canada, 2004.

[8] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[9] Knuth, D. Semantics of context-free languages. *Math. System Theory*, 2:127–145, 1968.

[10] http://www.macromedia.com/devnet/mx/coldfusion/articles/, 2004.

[11] Atsushi Ohori, Peter Buneman, and Val Breazu-Tannen. Database programming in Machiavelli–a polymorphic language with static type inference. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 46–57, 1989.

[12] Benjamin C. Pierce. *Types and Programming languages.* MIT Press, 2002.

[13] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal: Very Large Data Bases*, 10(4):334–350, ???? 2001.

[14] Reiter, R. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, 1978.

[15] Christian Soutou. Extracting n-ary relationships through database reverse engineering. In Bernhard Thalheim, editor, *Conceptual Modeling - ER'96, 15th International Conference on Conceptual Modeling, Cottbus, Germany, October 7-10, 1996, Proceedings*, volume 1157 of *Lecture Notes in Computer Science*, pages 392–405. Springer, 1996.

[16] http://www.databasejournal.com/, 2003.

[17] Dan Suciu. Typecheking for semistructureed data. In Giorgio Ghelli and Gösta Grahne, editors, *Database Programming Languages, 8th International Workshop, DBPL 2001, Frascati, Italy, September 2001, Revised Papers*, volume 2397 of *Lecture Notes in Computer Science*. Springer, 2001.

[18] http://www.tpc.org/tpch, 2004.

[19] http://www.microsoft.com/mind/0399/sql/sql.asp, 2004.

[20] http://en.wikipedia.org/wiki/, 2004.