TOWARDS UNDERSTANDING THE SCHEMA IN RELATIONAL DATABASES

ZHANG MEIHUI

Bachelor of Engineering

Harbin Institute of Technology, China

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

Zhang Meihui 29 July 2013

ACKNOWLEDGEMENT

This thesis would not have been possible without the guidance and support of many people during my PhD study. It is now my great pleasure to take this opportunity to thank them.

First and foremost, I would like to express my most profound gratitude to my supervisor, Prof. Beng Chin Ooi. Without him, I would not be able to complete my PhD program successfully. I was not from top university and did not come with very strong foundation and programming skills when I was admitted to the PhD program. I sincerely thank Prof. Ooi for his patience, guidance and support that helped me get through tough times and shape my research skills. I also thank him for offering me the opportunities to visit research labs and collaborate with accomplished researchers. It has been my great honor to be his student.

I would like to thank Dr. Divesh Srivastava, Dr. Cecilia M. Procopiuc, Dr. Marios Hadjieleftheriou and Dr. Hazem Elmeleegy for their valuable insights and advice during my internship at AT&T research lab. I had three great and productive summers with them. I would also like to thank Dr. Kaushik Chakrabarti for his guidance and suggestions during my spring internship at Microsoft Research.

I would like to thank my thesis advisory committee Prof. Stephane Bressan and Prof. Anthony K. H. Tung for their invaluable feedback at all stages of this thesis.

I would like to thank my other co-authors during my PhD study, especially Prof. Christian S. Jensen, Prof. Wang-Chiew Tan, Prof. Gao Cong, Prof. Hua

Lu, my seniors Ju Fan, Su Chen, Sai Wu, Dongxiang Zhang and Zhenjie Zhang for their conceptual and technical insights into my research work.

I thank all my colleagues in the database group. I would especially like to thank my nine-year roommate and my best friend Meiyu Lu. Thanks to you for helping me through all the hard times and accompanying me on our life journey.

Finally, I am deeply and forever indebted to my dear parents for their love, support and encouragement throughout my entire life.

CONTENTS

Acknowledgement **i**

Abstract **vii**

1 Introduction **1**

1.1 Brief Review of Relational Databases 2

1.1.1 Data Representation 2

1.1.2 Querying Relational Databaes 2

1.2 What Makes the Data Not Understandable 4

1.3 Uncovering the Hidden Relationships in the Data 5

1.3.1 Identification of Foreign Key Constraint 5

1.3.2 Discovery of Semantic Matching Attributes 7

1.3.3 Mining the Generating Query for SQL Answer Table . . 8

1.4 Objectives and Contributions 11

1.5 Thesis Organization 12

2 Literature Review **13**

2.1 Mining the Key-based Relationships 13

2.1.1 Discovery of Primary Keys 13

2.1.2 Discovery of Foreign Keys 15

2.2 Mining Semantic Relationships 16

2.2.1 Type-based Categorization 16

2.2.2 Schema Matching 16

2.3 Mining Query Structure 17
2.3.1 Query by Output 17
2.3.2 Synthesizing View Definitions 18
2.3.3 Keyword Search 18
2.3.4 Sample-Driven Schema Mapping 19
2.4 Summary 19

3 Foreign Key Discovery 20
3.1 Introduction 20
3.2 Preliminaries 25
3.3 Randomness 27
3.4 Overall Algorithm 33
3.5 Schema and Data Updates 36
3.6 Experimental Evaluation 37
3.6.1 Dataset Descriptions 38
3.6.2 EMD Computation 39
3.6.3 Overall Algorithm 40
3.6.4 Scalability 43
3.6.5 Column Names 43
3.6.6 Comparison With Alternatives 44
3.6.7 Inclusion Estimators 45
3.7 Summary 46

4 Attribute Discovery 47
4.1 Introduction 48
4.2 Preliminaries 50
4.2.1 Name Similarity 53
4.2.2 Value Similarity 53
4.2.3 Distribution Similarity 54
4.3 Attribute Discovery 55
4.3.1 Phase One: Computing Distribution Clusters 56
4.3.2 Phase two: Computing Attributes 60
4.4 Performance Considerations 66
4.5 Experimental Evaluation 68
4.5.1 Distribution Similarity 70
4.5.2 Attribute Discovery 71

4.6 Summary 76

5 Join Query Discovery **77**

5.1 Introduction 78

5.2 Preliminaries 80

5.2.1 Overview 80

5.2.2 Definitions 83

5.3 Query Generation 86

5.3.1 Step 1: Schema Exploration and Pruning 88

5.3.2 Step 2: Instance Trees and Star Centers 90

5.3.3 Step 3: Exploring Lattices 92

5.3.4 Step 4: Query Testing 93

5.4 Optimizations 95

5.4.1 Decreasing the depth d 96

5.4.2 Bounding TID list sizes 98

5.5 Experimental Evaluation 99

5.5.1 TPC-H Queries 99

5.5.2 Our Queries 101

5.5.3 Schema-Level Pruning 101

5.5.4 Instance-level Pruning 103

5.5.5 Optimizations: bounding TID size 105

5.5.6 Lattice Exploration and Testing 107

5.5.7 Optimizations: decreasing depth d 109

5.6 Summary 110

6 Conclusion and Future Work **111**

6.1 Conclusion 111

6.2 Future Work 112

Bibliography **114**

ABSTRACT

Database systems are adept at performing efficient computations over large datasets, as long as the queries are issued by users who understand the schema and can formulate their goals in the precise framework of SQL. However, the explosion of data over the past two decades has led to more and messier processing tasks than those envisioned by the creators of the SQL standard in the 1970s. One of the reasons for this departure from the classical model of user interaction with a DBMS is the fact that some crucial information is often unavailable.

In this thesis, we work towards designing solutions for relational databases to discover the information that is often undocumented and yet useful for people to understand and work with the data. More specifically, we first propose a general rule, termed *Randomness*, which effectively discovers meaningful foreign keys, including multi-column foreign keys. Second, we design a data oriented solution that identifies strong relationships between relational columns and clusters them into semantic attributes, i.e. the columns that have same or similar meaning are clustered together. Lastly, we provide a principled solution to discover complex generating queries for the cases where the user has the query answer and wants to find out the generating query for further investigation and analysis. Such information is invaluably helpful for database users to express their goals into SQL queries and generally to better understand and explore the data. We validate our proposed approaches via extensive experiments using real and benchmark databases.

LIST OF TABLES

3.1 Notation used throughout Chapter 3. 26

3.2 Datasets characteristics. 38

3.3 Foreign/primary keys according to schema specifications. 38

3.4 EMD accuracy for different quantile grid sizes; Diff=$EMD_{n,G_\ell} - EMD_{n,G_{2048}}$. 39

3.5 Number of candidate pairs that satisfy inclusion; SC=single-column, MC=multi-column. 40

3.6 False negatives in TPC-E (A=Active, B=Completed, C=Canceled, D=Pending, E=Submitted). 43

3.7 Results after eliminating non-matching column names. 45

4.1 Notation used throughout Chapter 4. 52

4.2 Datasets statistics. 69

4.3 Description of materialized views. 69

4.4 Attributes that contain horizontally partitioned columns in TPC-H. 71

4.5 Accuracy results on TPC-H, IMDB and DBLP for different thresholds θ; m' true number of attributes; m attributes in our solution; P is precision; R is recall. 73

4.6 Attributes that are incorrectly clustered together in TPC-H for $\theta = 0.12$. 74

5.1 Experiment parameters and settings 100

5.2 Results on TPC-H queries (grouped by number of joins) 100
5.3 TPC-H query set. (The projection tables are underlined.) 102
5.4 Characteristics of Θ_1 and Θ_2. 102
5.5 Testing time of query graphs for Q2. 108
5.6 Tested candidate graphs for Q3 (note: G1=G9=G13). 109
5.7 Number of graphs and testing time for Q3, at $d = 3$. 109

LIST OF FIGURES

1.1 Excerpt of the schema graph of the `UNIVERSITY` database. . . . 3

1.2 A subset of the `UNIVERSITY` database schema with three foreign keys. 6

1.3 The semantic matching attributes of the `UNIVERSITY` database. 8

1.4 Examples of join queries over the `UNIVERSITY` database. . . . 10

3.1 A small subset of the TPC-E schema with one multi-column and several single-column foreign keys. 21

3.2 Constructing a Bottom-k sketch. 27

3.3 A good foreign key F is a set of random values from the primary key. Column F' fails the randomness test. 28

3.4 A column containing numeric values might falsely appear to be a random sample of a primary key based on lexicographic sorting of values. 28

3.5 The Wilcoxon test: 1. Sort values in multi-set $F \cup P$; 2. Assign ranks; 3. Compute the rank-sum of values in F (13.5 in this example). 29

3.6 EMD quantifies the amount of work required to convert one set of values into another. 30

3.7 Constructing a 2-dimensional 4-quantile histogram for primary key P. 32

3.8 Utility measures on TPC-H, Wikipedia and IMDB. 41

3.9 Utility measures on TPC-E using the golden standard and extended constraints. 42

3.10 Scalability results. 44

3.11 Accuracy of bottom-k estimators for the inclusion coefficient, as a function of k. 46

4.1 Excerpt of the TPC-H schema. 50

4.2 Attributes in TPC-H example, which contains three base tables and two materialized views of CUSTOMER table. 51

4.3 Data distribution histograms of two examples from TPC-H. . . 54

4.4 EMD plot of two examples in TPC-H. 57

4.5 Distribution clusters of TPC-H example. 60

4.6 A possible attribute graph of distribution cluster $\mathbf{DC}_1$. 63

4.7 Attributes discovered in the attribute graph of distribution cluster $\mathbf{DC}_1$. 65

4.8 Another possible attribute graph of distribution cluster $\mathbf{DC}_1$. . . 65

4.9 Distribution histograms of EMD values between all pairs of columns in the same attribute for TPC-H and DBLP. 70

4.10 Distribution histograms of Jaccard values between all pairs of columns in the same attribute for TPC-H and DBLP. 72

4.11 Accuracy results on TPC-H, IMDB and DBLP for varying thresholds θ. 73

4.12 An attribute sub-graph of TPC-H for varying thresholds θ. . . . 75

5.1 The TPC-H schema and two **R**unning example **Q**ueries over it. 79

5.2 Example candidate graphs for RQ2: naive approach. 81

5.3 Illustration of our graph characterization result. 82

5.4 Example of lattice. An edge corresponds to a merge step. 85

5.5 Computing the query in Figure 5.1(c) via Algorithm 5.1. 87

5.6 TPC-H instance (only relevant tables and columns are shown; column names are abbreviated). 89

5.7 Algorithmic steps for table *Out* = "`SELECT PS.suppcost, L.shipdate, O.orderdate FROM PartSupplier as PS, PartSupplier as PS1, Part as P, Supplier as S, LineItem as L, LineItem as L1, Orders as O WHERE PS1.skey = S.skey and S.skey = PS.skey and PS1.pkey = P.pkey and P.pkey = L.pkey and PS1.pkey = L1.pkey and PS1.skey = L1.skey and L1.okey = O.okey`" (it-s graph is isomorphic to $Star2$). 89

5.8 Proof of Theorem 5.1: (a) A query graph Q (black edges) and its directed version Q_d (green edges); (b) Modified Euler tour E_m; (c) Discovering a star whose lattice contains Q. 94

5.9 Computing the query in Figure 5.1(b) via Algorithm 5.1: (a)no optimizations, $d = 5$; (b) generalized stars, $d = 3$; (c) intersection, $d = 2$. 96

5.10 Effects of schema-level pruning for Q1. 103

5.11 Instance-level pruning for Q4, as a function of $|\Theta|$. 104

5.12 Bounding TID sizes. 106

5.13 Lattice for Q2; $d = 1$. 107

5.14 Lattices for Q3; $d = 3$. 107

5.15 Effects of intersection for Q5 and Q6. 110

CHAPTER 1

Introduction

In the age of information explosion, people are facing technical difficulty in organizing, storing and managing the data. For that reason, relational database systems are developed to provide an effective tool that simplifies the above tasks and assists people in extracting useful information in a timely fashion. However, as the databases increase, both in size and number, it is getting more and more difficult to understand and work with the data.

One of the reasons for this consequence is the fact that some crucial information, such as the database structure, integrity constraints and view definitions, are often unavailable due to insufficient (or missing) documentation or performance and security concerns. When this happens in enterprise databases, which easily contain hundreds or thousands of inter-linked tables, even domain expert users will have a difficult time understanding the data in order to express their goal in the form of SQL queries. Therefore, to ensure that the databases are as useful and helpful as they ought to be, automatic tools and methodologies are required to help people understand the data in relational databases.

In this chapter, we first briefly review the data representation and exploration in relational databases, and attempt to analyse the reasons why relational data are often not easy to interact with. Subsequently, we overview our practical solutions to assist users in understanding the data by means of discovering useful information from the data. Finally, we summarize the objectives of this research work and outline the thesis organization.

1.1 Brief Review of Relational Databases

A relational database is a collection of data items organized based on *relational model* [20], which was first introduced by Edgar F. Codd of IBM Research in 1970. Due to its simplicity and mathematical foundation, the relational model has attracted immediate attention and become the predominant data model in storing and managing the data. It forms the basis for today's commercial database management systems (DBMSs) including IBM's DB2 and Informix, Microsoft's SQLServer and Access, Oracle and Sybase. In addition, several open source systems, such as MySQL and PostgreSQL, are implemented based on the relational model as well.

1.1.1 Data Representation

The relational model represents the database as a collection of relations (or tables), where each relation is a table with rows (or tuples, or records) and columns (or attributes, or fields). The *relational schema* specifies various properties of the tables in the database, e.g., the table names, the columns within each table, the type of data contained in each column, indices, constraints etc. [50] One of the most important constraints is the *foreign key* constraint that defines the *referential* relationship between columns of different tables. Specifically, the foreign key column in the referencing table must be a subset of the primary key column in the referenced table. The *schema graph* is often used to visualize the structure of a database, and defined as follows: the nodes correspond to the tables and the edges to the foreign/primary key (fk/pk) relationships.

We take a portion of a `UNIVERSITY` database as an example, which contains six tables, `STUDENT`, `STAFF`, `DEPARTMENT`, `MODULE`, `PREREQUISITE` and `GRADE_REPORT`. As shown in Figure 1.1, the schema graph presents the tables, columns in each table, data type and the foreign/primary key relationships between the columns.

1.1.2 Querying Relational Databaes

SQL (Structured Query Language) is a standard language designed for accessing and manipulating the data held in relational databases. SQL is comprehensive:

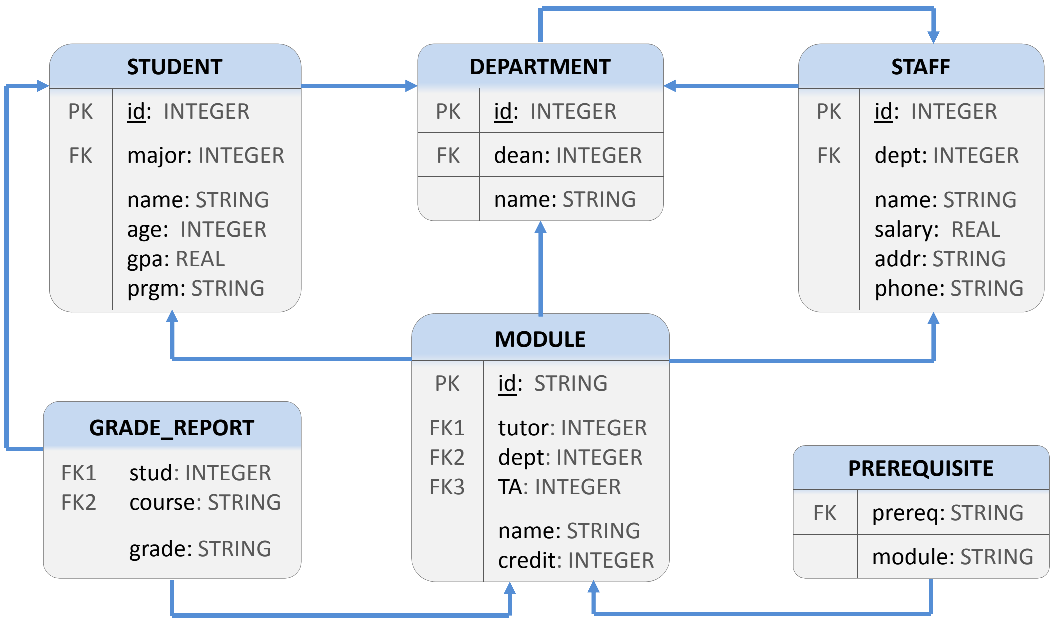


Figure 1.1: Excerpt of the schema graph of the `UNIVERSITY` database.

it has statements for specifying the data definitions, for defining integrity constraints, for creating views on the database and for altering the schema and the data etc.

The most common operation in SQL is the query, which is the way to retrieving information from a database. Queries in SQL can be very complex. The basic form of SQL queries is a `SELECT-FROM-WHERE` structure, where the `SELECT` clause specifies the *projection attributes* (the attributes whose values are to be retrieved), the `FROM` clause lists the tables required to process the query and the `WHERE` clause specifies the *selection conditions* and the *join conditions* (if any). More complex queries contain aggregates, arithmetic expressions, nested queries etc. by means of `GROUPBY`, `EXISTS` and other operators. A query that involves only selection and join conditions plus projection attributes is known as a *Select-Project-Join* (SPJ) query. The next example is a SPJ query with two projection attributes, one selection condition and two join conditions over the `UNIVERSITY` database (see Figure 1.1 for the schema graph).

QUERY 1: *Retrieve the name and address of all staff who work for the 'Computer Science' department and have teaching experience.*

```
SELECT  STAFF.name, STAFF.addr
FROM    STAFF, DEPARTMENT, MODULE
WHERE   DEPARTMENT.name = 'Computer Science'
AND     STAFF.dept = DEPARTMENT.id
AND     STAFF.id = MODULE.tutor
```

1.2 What Makes the Data Not Understandable

Database systems are adept at managing large datasets and performing efficient computations as long as the queries are issued by the users who understand the schema and are familiar with the data. Nevertheless, understanding the data in complex databases is sometimes rather challenging.

First of all, the schema information, which is the basis for users to understand the database structure, is often unavailable. Sometimes, this is the result of poorly documented legacy databases [24, 25]. The following was reported in a real case study of the Holy Cow Corp. in [24].

> "*The documented metadata was a microscopic part of the metadata needed to correctly interpret the data.* "
>
> "*Furthermore, the taskforce found that there were many changes made daily without documentation or notification.*"

Sometimes it may even be the deliberate decision of the database administrator to not specify integrity constraints (e.g., foreign/primary key relationships) for performance considerations. In other cases, it is not feasible to specify those constrains due to the data inconsistencies that may arise from data integration or database evolution. However, it is nearly impossible to extract useful information through SQL queries without understanding the schema. For example, one has to know the foreign/primary key relationships between STAFF, DEPARTMENT and MODULE to form the join conditions in the SQL of Query 1. Indeed, developing algorithms for the automatic *discovery of schema information* has attracted much interest in research community and is an ongoing area of research.

In a more complex scenario, the desired information may be spread across multiple database sources, each with its own schema. In order to issue appropriate SQL queries and extract useful information out of the relevant sources, one has to understand each local schema as well as the global structure. This requires the identification of semantic correspondence between different database instances. Finding such matching relationships, also known as *schema matching*, is not only a crucial step in exploring and querying the databases but also a fundamental task in data integration process.

In practice, many database users could share database instances. They compute an SQL answer and store it into a view or a temporary table, then share it without annotating it with the generating query. To make the matters worse, even the table creator himself might forget the generating query after a while if it is not documented properly. However, knowing how tables are generated is very useful. For instance, someone may notice inconsistencies in the output and want to investigate, or they may want to generate a slightly different output for further analysis. Awareness of the generating query of the output tables can also prevent creating redundant tables.

Finally, the explosion of data over the past two decades aggravates the above problems. As the databases grow more massive and the schemata become more complex, understanding and exploring the databases becomes extremely challenging. It is thus imperative to develop automatic tools that simplify the process of understanding the relational data.

1.3 Uncovering the Hidden Relationships in the Data

In this thesis, we aim to design new approaches to analyze database instances to efficiently and accurately discover information that is useful for assisting users in understanding and exploring the relational databases. In view of the practical scenarios that we discussed in the previous section, we tackle the task from the following three perspectives.

1.3.1 Identification of Foreign Key Constraint

As we have seen in earlier discussion, knowledge of database schema enables richer queries (e.g., joins) and more sophisticated data analysis. For that reason, we first bring our attention to one of the most important schema elements, the foreign key constraint.

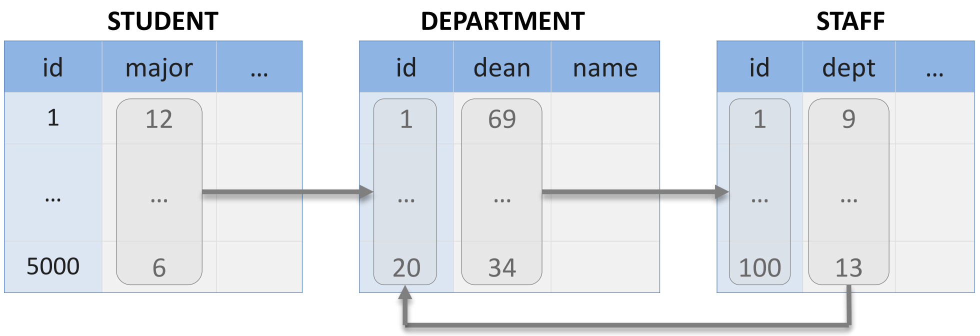


Figure 1.2: A subset of the `UNIVERSITY` database schema with three foreign keys.

We propose a novel approach for efficiently and accurately discovering meaningful foreign keys in relational databases, including multi-column foreign keys, which have not been considered by pervious studies. Even for single-column foreign keys, existing work concentrates mainly on identifying inclusion dependencies (the detailed review will be provided in Chapter 2). This is simply because the containment relationship between the primary/foreign key column is the only formal requirement for specifying the foreign key constraint. However, checking only for inclusion can easily lead to a large number of false positives. Consider the columns in the `UNIVERSITY` database in Figure 1.2 as an example. There are six columns in the figure containing integers ranging in different intervals. While `STUDENT.id` fully contains the other five integer columns, none of them is in fact related to `STUDENT.id`. Thus, a simple inclusion test would incorrectly report something like `STUDENT.id` and `STAFF.dept` is in a foreign/primary key relationship. This scenario arises frequently in real-world databases since the auto-increment fields are commonly used in practice.

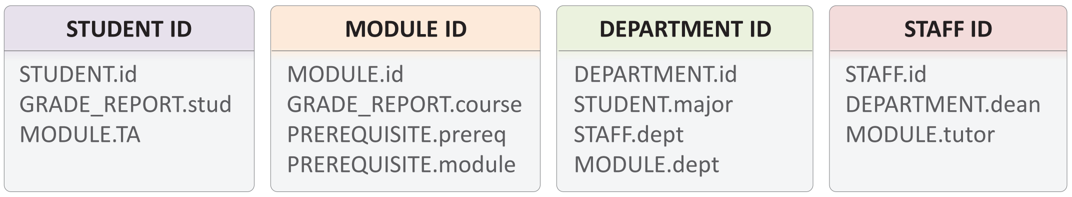
However, our approach can effectively reduce the number of false positives produced by the inclusion test. Regarding to the example in Figure 1.2, only the three true foreign keys, i.e. `STUDENT.major` → `DEPARTMENT.id`, `DEPARTMENT.dean` → `STAFF.id` and `STAFF.dept` → `DEPARTMENT.id` will be reported as meaningful foreign keys in the output of our approach.

Our approach is based on the key insight that in most cases the values in a foreign key column form a nearly uniform random sample of the values in the primary key column. In other words, it is highly unlikely that a database instance is designed such that a foreign key column is a biased sample of the respective primary key, e.g., a prefix or a suffix in the ranked order. Even if this is the case at the first time the database instance is populated, for dynamic databases the distribution of the values in foreign/primary key is expected to change over time, and eventually such bias should be eliminated. Based on this observation, we conjecture the closer a column F is to a uniform random sample of a primary key column P, the higher the likelihood that the (F, P) pair is a meaningful foreign/primary key constraint. We thus propose a novel foreign key discovery rule, termed *Randomness*, that uses the data distribution (previous works apply simple heuristic rules such as column names and min/max values to prune the false positives produced by the inclusion test) to measure the randomness of a candidate foreign key column with respect to a specific primary key column. This way, we can quantify the likelihood that a pair of columns that satisfy inclusion is a useful foreign/primary key constraint. Applying the randomness rule to the example in Figure 1.2, it is clear that unrelated column pairs like `STUDENT.id` and `DEPARTMENT.id` can be effectively eliminated from the candidates which have passed the inclusion test, since the subset column (`DEPARTMENT.id`) forms a biased sample (prefix) of the other one.

1.3.2 Discovery of Semantic Matching Attributes

The second practical problem we address is automatic discovery of semantic matching attributes in relational databases. We have seen earlier that the data in relational databases are described in the form of relational schema. While the schema provides us a way to specify various properties of the data contained in the databases, including the data type for each column and the foreign/primary key relationships between columns, it has certain limitations in practice. In particular, one cannot accurately name the columns that can be "semantically" joined/unioned (other than the foreign/primary keys) by just looking at the schema only and not fully understanding the data. Clearly, the columns that are in the same primitive data type are very likely to be unrelated, e.g., `STUDENT.gpa` and `STAFF.salary` are both real numbers. To make the matter worse, the foreign keys are sometimes not specified in the schema for various reasons (see discussion in Section 1.2).

In this thesis, we design an automatic, unsupervised and purely data oriented approach for clustering relational columns into semantic matching at-



STUDENT ID	MODULE ID	DEPARTMENT ID	STAFF ID
STUDENT.id	MODULE.id	DEPARTMENT.id	STAFF.id
GRADE_REPORT.stud	GRADE_REPORT.course	STUDENT.major	DEPARTMENT.dean
MODULE.TA	PREREQUISITE.prereq	STAFF.dept	MODULE.tutor
	PREREQUISITE.module	MODULE.dept	

Figure 1.3: The semantic matching attributes of the `UNIVERSITY` database.

tributes. We do not rely on the existence of any external knowledge, e.g., foreign/primary key relationships, column names etc. As an illustration, we show the clustering of the columns in the `UNIVERSITY` database (see Figure 1.1) in Figure 1.3. (The columns that are absent in Figure 1.3 do not have matching columns and form a cluster on their own.) We see from the figure that the following types of columns are clustered together: (1) the foreign/primary key, e.g., `GRADE_REPORT.course` and `MODULE.id`, (2) the foreign keys that refer to the same primary key, e.g., `GRADE_REPORT.course` and `PREREQUISITE.prereq`, (3) even the columns that have no explicit relationship but semantically equivalent, e.g., `GRADE_REPORT.course` and `PREREQUISITE.module`. Two more types are possible when views exist in the database instance: (1) the column in the view table and its corresponding column in the base table, (2) the columns (in view tables) that are from the same corresponding column in the base table.

Our approach provides a robust tool that identifies all of the above types of relationships (our first work has studied the type 1 but not the rest of them) and reports a clustering of columns into semantic matching attributes. Apparently, such information is invaluably helpful for database users to formulate their join queries and generally, to better understand and work with the data. Our work can also be used as a valuable addition to the existing techniques for designing automated data integration and schema mapping tools.

1.3.3 Mining the Generating Query for SQL Answer Table

The third problem we focus on is the following inverse problem: suppose that a user already has the output table of an SQL query and the source database (or multiple database instances), and intents to discover the generating query that produces the table.

Note that for most of the queries (if not all), there exist *instance-equivalent queries* [56], i.e. the queries that produce equivalent output table with respect to a database instance. By default, our approach returns the instance-equivalent query with the smallest complexity assuming that a complexity measure (e.g., the number of joins/tables etc.) is pre-defined over the queries. A few variants of the problem are as well considered in our approach. For example, one may wish to generate a query that outputs a superset of the given SQL answer. In other cases, one may want to know all of the instance-equivalent queries.

As discussed previously, this problem has numerous potential applications, both by itself, and as a building block for other problems. For instance, in the area of database exploration and analysis the ability to discover the query for SQL answer is very useful, especially when the required documentation and metadata are incomplete, missing or nowhere accessible. In addition, deriving the instance-equivalent queries could aid in uncovering the hidden relationships that are interesting to the users but unknown a priori. As an example, one might be surprised to find that the students who did well in a particular module are in fact the ones who come from a particular department (the example queries are shown below in QUERY 2 and QUERY 3) through the instance-equivalent queries.

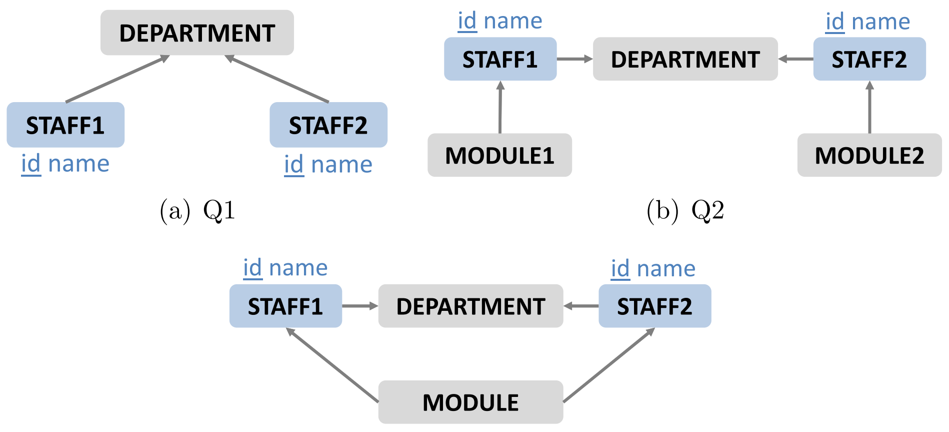
QUERY 2: *Retrieve the id and name of all students who got 'A$^+$' grade in 'Decision Making' module.*

```
SELECT  STUDENT.id, STUDENT.name
FROM    STUDENT, MODULE, GRADE_REPORT
WHERE   STUDENT.id = GRADE_REPORT.stud
AND     MODULE.id = GRADE_REPORT.course
AND     MODULE.name = 'Decision Making'
AND     GRADE_REPORT.grade = 'A+'
```

QUERY 3: *Retrieve the id and name of all students who are from 'Computer Science' department in 'Decision Making' module.*

However, solving this problem is non-trivial. First of all, the number of potential candidate queries is usually super-exponential to the query graph size, especially for the case of cyclic schema graph. Thus, simple solutions like brute-force approaches that enumerate and test all possible queries (up to some complexity) are certainly not suitable.

```
SELECT  STUDENT.id, STUDENT.name
FROM    STUDENT, MODULE, GRADE_REPORT, DEPARTMENT
WHERE   STUDENT.id = GRADE_REPORT.stud
AND     MODULE.id = GRADE_REPORT.course
AND     STUDENT.dept = DEPARTMENT.id
AND     MODULE.name = 'Decision Making'
AND     DEPARTMENT.name = 'Computer Science'
```



(a) Q1 (b) Q2 (c) Q3

Figure 1.4: Examples of join queries over the UNIVERSITY database.

Consider the following queries (the query graphs are illustrated in Figure 1.4 where the projection tables are with the projection columns next to them):

Q1: Find all pairs of staff members who work in the same department.

Q2: Find all pairs of staff members who work in the same department and have teaching experience.

Q3: Find all pairs of staff members who work in the same department and teach (taught) the same module.

Clearly, the outputs of these three queries are overlapping but not identical. Effectively distinguishing between the queries that have similar results become another challenge.

We have a crucial insight that any join query can be characterized by the combination of a simple structure, called a *star*, and a series of merge steps over the stars. Based on the observation, we propose an efficient approach that uses the star construct to discovers arbitrary join queries.

1.4 Objectives and Contributions

To summarize, the following specific problems exist in relational databases in reality:

- The foreign/primary key relationship, one of the most important constraints in a database, is often not known to database users for various reasons. Without the information of the foreign keys, performing data exploration and analysis become rather challenging, especially for the databases with complex schema.
- Even when the database schema is available (but not additional helpful documentation), the schema itself is inadequate for users to fully understand the data in terms of the semantically joinable columns.
- Unless the original query is somewhere properly documented, it is of great difficulty for database users to figure out the query that generates an output table and to further investigate or utilize it.

In this thesis, we work towards designing solutions for relational databases to discover the information that is often undocumented and yet useful for people to understand and work with the data. In particular, we seek to achieve the following specific objectives:

- To design an effective approach to discover foreign key constraints. The approach should be able to reduce a large number of false positives produced by the inclusion checking in order to make the identification of useful relationships feasible.
- To provide a solution to identify the strong relationships between columns in terms of the semantic equivalence, i.e. to identify the strongly connected columns that have same or similar meaning within the context of certain domain.
- To study the problem of discover the query for SQL answer tables and design a principled solution. The solution should be able to efficiently prune out a large number of false candidates and scale to large databases and complex queries.

The main contributions of this thesis are summarized as follows: First, we propose a novel rule, termed *Randomness*, which can effectively discovers meaningful foreign keys, including multi-column foreign keys that have not been considered by previous work. Second, we introduce a robust and data oriented solution that use statistical measures to cluster relational columns into semantic attributes. Finally, we propose an efficient method for discovering arbitrary join queries (in contract, related prior work imposes restrictions on the structure of the query). We design several optimizations that significantly reduce the running time, making our method scalable.

1.5 Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 discusses the related works.

Chapter 3 addresses the problem of discovering single and multi-column foreign keys. A novel distance measure is defined to quantify the likelihood that a pair of columns which satisfy inclusion is a meaningful foreign/primary key constraint.

Chapter 4 studies the problem of identifying semantic matching attributes from the data. A two-phase approach is presented to cluster relational columns into attributes based on their semantic equivalence.

Chapter 5 introduces a principled approach to the problem of discovering complex join queries for SQL answer tabless. An efficient algorithm is proposed to efficiently explore the set of candidates and quickly prune out a large number of infeasible queries.

Chapter 6 concludes the thesis and discusses possible future work.

CHAPTER 2

Literature Review

There have been a lot of research work proposed to assist users in understanding and interacting with database systems from various aspects. In this chapter, we review those that are closely related to this thesis. In particular, we first discuss the existing techniques on discovering the key-based relationships. We next introduce current solutions of clustering relational columns. We also briefly review the schema matching techniques. Finally, we discuss the work on mining query structures and analyse the limitations of the prior methods.

2.1 Mining the Key-based Relationships

Understanding the structure and relationships in databases is important and yet a difficult task especially for large industrial-scale databases with poor documentation. Tools and techniques have been proposed to make sense of the relational data from various aspects. For example, the tool developed by AT&T, called *Bellman* [25], collects compact statistical summaries of the database contents and uses these summaries to mine the database structure. In this section, we mainly review the related work on discovering primary and foreign keys.

2.1.1 Discovery of Primary Keys

A primary key is a special case of a *functional dependency* [19], since it trivially determines the values of all columns in the same table. A large body of work has concentrated on exact and approximate functional dependencies.

Functional Dependencies

Functional dependencies (FDs) are relationships between attributes of a relation: Given a relation R, a set of attributes $X \subseteq R$ is said to functionally determine another set of attributes $Y \subseteq R$, written as $X \to Y$, if and only if tuples that agree in all attributes of X must agree in all attributes of Y. A FD is said to be minimal if Y is not functionally dependent on any proper subset of X. Algorithms for computing minimal functional dependencies are proposed in [34, 38, 57].

Huhtala et al. proposed and implemented TANE system [34, 33] for finding both functional and approximate dependencies from large databases. Their approach is based on the idea of partitioning the set of rows with respect to their attribute values. The use of partitions allows them to easily identify erroneous/exceptional values and quickly discover approximate functional dependencies. Dep-Miner [38] takes in stripped partition databases as input. A stripped partition database encompasses stripped partitions for each attribute. A stripped partition is no difference with the partition in TANE, except that the partition must have a size greater than one. Using such partitions, agree sets and maximal sets are then generated. Finally, FDs according to the maximal sets are found. Both TANE and Dep-Miner search for FDs in a breadth-first or levelwise manner. FastFDs [57] differs from Dep-Miner only in that FastFDs uses a depth-first search strategy.

Primary Keys

Little work has tackled the discovery of primary keys in particular, especially multi-column primary keys (a.k.a composite keys). Even for single-column keys current algorithms use a brute-force approach. The Bellman system [25] implemented a levelwise key finding algorithm simlar to TANE [33]. The state of the art for efficient, automatic discovery of single and multi-column primary keys is GORDIAN [54]. GORDIAN formulates the problem as a cube computation [29] that corresponds to the computation of the entity counts of all possible column projections. The algorithm first discovers all non-keys, since a non-key can usually be identified after looking at only a small subset of the entities.

2.1.2 Discovery of Foreign Keys

Inclusion Dependencies

Surprisingly, very little work has dealt with discovery of foreign keys. Most work focuses on computing inclusion dependencies only [41, 17, 10, 42, 37].

Bauckmann et al. [10] proposed Spider for efficiently detecting single-column inclusion dependencies. The algorithm first sorts the distinct values in all columns and then uses a parallel merge-sort like algorithm to compute all inclusions simultaneously. Spider computes inclusions exactly, but the cost is super-linear to the size of the data. The algorithm is also based on parallelization, where all columns are scanned concurrently.

A similar approach was proposed by Marchi et al. [41], using a linear pass over the data to compute an inverted index over each data type (e.g., strings, floats, integers). Subsequent passes over the index can discover single/multi-column inclusions.

Marchi and Petit [42] proposed a hybrid technique based on association rule mining to find low-dimensional inclusions and an optimistic exploration of high-dimensional inclusions using clique-finding. Koeller and Rundensteiner [37] utilize clique-finding for discovering high-dimensional inclusions. Partial inclusion is not addressed in these works.

Dasu et al. [25] proposed using minhash sketches to find potential associations between columns (or sets of columns) as a function of Jaccard coefficient. However, Jaccard is not a good indicator of inclusion coefficient when the set sizes differ substantially.

Foreign Keys

Inclusion is not a sufficient condition for foreign keys, resulting in a large number of spurious keys. Rostin et al. [51] introduced a machine learning approach for discovering foreign keys that is based not only on inclusion, but on a variety of other properties of good foreign keys. The authors use the Spider algorithm to discover all inclusion dependencies, and use SQL queries to evaluate various properties on the data, resulting in a very expensive pre-processing step. Most importantly, the algorithm requires a learning step, which implies the availability of datasets with known foreign/primary keys. The quality of the training dataset affects performance significantly. Finally, multi-column foreign keys are not addressed in that work.

Lopes et al. [39] proposed a query workload based approach to discover foreign key relationships based on the assumption that SQL join queries use foreign/primary keys. This approach is based on the availability of a query workload.

2.2 Mining Semantic Relationships

From a data analysis perspective, knowing the semantic relationships between relational columns is a necessary step to understand and process the data. Previous work tangentially related to discovering semantic relationships is that on quickly identifying columns that contain similar values. A number of statistical summaries have been developed for that purpose, including min-hash signatures [16], and locality sensitive hashing [28]. These techniques cannot be used for discovering semantic relationships, since they only capture the data intersection relationships between columns.

2.2.1 Type-based Categorization

In the context of relational databases, there has been little work that concentrates on classifying columns into semantic clusters. The only previous work that we are aware of is by Ahmadi et al. [7], that utilizes q-gram based signatures to capture column type information based on formatting characteristics of data values (for example the presence of '@' in email addresses). The technique builds signatures based on the most popular q-grams within each column and clusters columns according to basic data types, like email addresses, telephone numbers, etc. In that respect, the goal of this work is orthogonal to ours: It tries to categorize columns into generic data types.

2.2.2 Schema Matching

Schema matching is the process of identifying that two columns are semantically related. Automating the process of schema matching has been one of the fundamental tasks of data integration. Related work from the field of schema matching has concentrated on three major themes.

- The first is semantic matching that uses information provided only by the schema and not from particular data instances.
- The second is syntactic schema matching that uses the actual data instances.
- The third uses external information, like thesauri, standard schemas, and past mappings.

Most solutions use hybrid approaches that cover all three themes. Rahm and Bernstein [49] conducted a survey on schema matching techniques. Current approaches use string-based comparisons (prefix/suffix tests, edit distance, etc.), value ranges, min/max similarity, and mutual information based on q-gram distributions [35, 26, 27, 40, 43].

2.3 Mining Query Structure

There have been a lot of research work that aim to mine the query structure for a particular SQL answer table. Formally, they address the following reverse engineering problem: Given an output table Out, discover the query Q that generates Out. All related results we are aware of impose restrictions on the structure of the query graph Q. Thus, they only explore a subspace of possible solutions. We mention the specific restrictions for each case as below.

2.3.1 Query by Output

The algorithm proposed in [56], dubbed TALOS, focuses on the selection conditions of an SPJ query: given a query graph Q, it computes its output $Out(Q)$, then discovers the best selection conditions that, when applied to $Out(Q)$, generate table Out. The graph of Q is assumed to be a subgraph of the schema graph, and computed by exhaustive enumeration. However, many queries are not subgraphs of the schema graph; e.g., the queries in Figure 1.4 (see Figure 1.1 for its schema graph). Moreover, exhaustive enumeration is either infeasible or impractical.

2.3.2 Synthesizing View Definitions

The work by Das Sarma et al. [52] has a problem statement that Out is a view instance and the goal is to find the view definition. They consider different metrics for ordering queries:

- Family of queries: a restriction that forces Q to be from a specific family of queries, e.g., single predicates or conjunctive queries.
- Level of approximation: a relaxation that allows the output of Q is close to but not exactly Out.
- Succinctness: a factor that measures the complexity of the return query Q.

However, they only consider views derived by (different families of) selection predicates over a specified single table. In other works, the queries that involves joins are not addressed in this work.

2.3.3 Keyword Search

In the area of keyword search over databases, table Out has only one tuple whose fields consist of the specified keywords. Some of the prior work [6, 14] computes a SQL query that generates a superset of Out, although the majority of results [12, 31, 32, 48] connect the keywords via graphs at tuple level. For our problem setup, we would have to issue a separate keyword search query for each tuple in Out. However, we may get back different SQL queries for different tuples, or else a single query that generates a superset of Out. Moreover, the query is usually a tree at tuple level [6, 12, 31, 32], whose leaves contain at least one keyword. There are however many counterexamples. For instance, at tuple level, Q3 in Figure 1.4(c) is not a tree, and Q2 in Figure 1.4(b) does not contain keywords in its leaves.

The approach in [48] discovers more complex tuple graphs, dubbed *communities*: they are superpositions of all depth-d trees whose leaves contain the keywords. However, a (tuple-level) community may lead to multiple SQL queries, and its model is still too restrictive for certain generating queries.

2.3.4 Sample-Driven Schema Mapping

In schema mapping with output samples [46], we are given the source schema(s), source table(s) and table *Out*, which consists of a small number of tuples from a table in the destination schema. The SQL query Q usually generates a superset of *Out*; once computed, it is included in the schema mapping. In [46] the query graph is assumed to be a tree (at instance level), which is a limitation in practical settings.

2.4 Summary

In this chapter, we have reviewed related work on discovering foreign keys. Most of the work focus on identifying the inclusion dependencies between relational columns [41, 17, 10, 42, 37], which however may yield a large number of spurious foreign keys. The recent machine learning approach [51] fails to discover the multi-column foreign keys. Various techniques [7, 35, 26, 27, 40, 43] have been proposed to mine the relationships between relational data columns. However, existing data driven approaches have not used any distributional information to discover relationships between columns, apart from simple statistics. Finally, we have reviewed related prior work on mining the query structures [56, 52, 6, 12, 31, 32, 46]. However, they all impose conditions on the structure of the query graph Q, and thus they have limitations in practical settings.

CHAPTER 3

Foreign Key Discovery

A foreign/primary key relationship between relational tables is one of the most important constraints in a database. From a data analysis perspective, discovering foreign keys is a crucial step in understanding and working with the data. Nevertheless, more often than not, foreign key constraints are not specified in the data, for various reasons; e.g., some associations are not known to designers but are inherent in the data, while others become invalid due to data inconsistencies. In this chapter, we propose a robust algorithm for discovering single-column and multi-column foreign keys. Previous work concentrated mostly on discovering single-column foreign keys using a variety of rules, like inclusion dependencies, column names, and minimum/maximum values. In this chapter, we first propose a general rule, termed *Randomness*, that subsumes a variety of other rules. We then develop efficient approximation algorithms for evaluating randomness, using only two passes over the data. Finally, we validate our approach via extensive experiments using real and synthetic datasets.

3.1 Introduction

A foreign/primary key relationship between relational tables is one of the most important constraints in a database. From a data analysis perspective, discovering foreign keys is a crucial step in understanding and working with the data. For that reason, database systems allow the explicit specification of foreign key

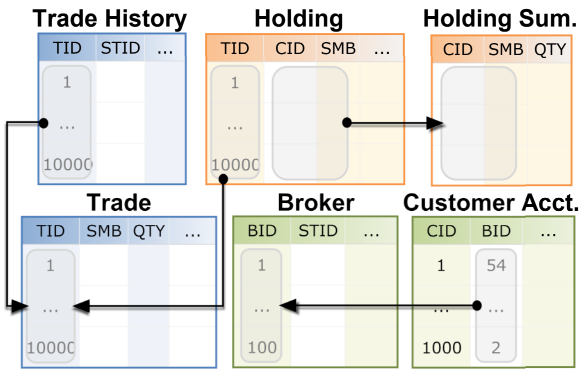


Figure 3.1: A small subset of the TPC-E schema with one multi-column and several single-column foreign keys.

constraints in the database schema. Nevertheless, in practice, database designers frequently fail to specify such constraints for various reasons, including: they are not aware of implicit relationships inherent in the data; such relationships might hold across multiple databases; it is not feasible to specify the constraints due to data inconsistencies (e.g., those arising from data integration or from database evolution over time); or because of performance considerations. When this happens in enterprise databases, which often contain hundreds of tables, thousands of columns and insufficient (or missing) documentation, even expert users have a difficult time identifying foreign key constraints.

In this chapter, we propose a novel approach for discovering foreign/primary key (fk/pk) relationships between single or multiple columns in relational databases. Surprisingly, little previous work deals with the case of discovering multi-column foreign keys [41]. Even for single-column keys, existing work is limited and focuses mainly on identifying inclusion dependencies, since the only formal requirement for specifying a foreign key constraint is that the foreign key be a subset of the primary key [41, 10]. However, checking only for inclusion can lead to a large number of false positives.

For example, Figure 3.1 shows a portion of the benchmark TPC-E schema, which represents a stock transaction system. It has information about customer accounts, companies, brokers, stock trades, etc. Column Trade.TID contains all integers in the interval $[1, 10000]$, while column Broker.BID, which is unrelated to TID, contains all integers in $[1, 100]$. A simple inclusion test would incorrectly report (Broker.BID, Trade.TID) as a foreign/primary key pair. This scenario arises frequently in practice because of auto-increment fields. Of course, one could adapt the test so that it discards pairs in which one column is a consecutive subset (e.g., a prefix or a suffix) of the other. However, that is not sufficient. Notice that the values in column Customer Account.BID, which is a foreign key of column Broker.BID, are a *random subset of a prefix* of Trade.TID. Hence, the inclusion test adapted as above would still incorrectly report (Customer Account.BID, Trade.TID) as a foreign/primary key pair. To complicate matters further, this problem is not limited to numerical attributes. It arises with date-time fields that may contain consecutive values, or even alphanumeric fields composed of letters followed by a number (e.g., A-1, A-2). The same is true for multi-column keys. For example, Holding.(CID, SMB) is a two-column foreign key of Holding Summary.(CID, SMB). However, Broker.(BID, STID) is not a valid foreign key of Trade History.(TID, STID), even though column-wise inclusion is satisfied.

Reducing the number of false positives is a critical requirement in order to make the identification of useful relationships feasible. As we show in the experimental section the number of false positives (i.e., pairs of columns that satisfy inclusion but are not valid fk/pk constraints) can be in the order of hundreds. Even for domain experts the task of sifting through and manually validating candidates is overwhelming. Previous work has proposed heuristic rules to reduce the number of false positives by identifying important properties that a good foreign key should satisfy. A comprehensive list of such properties, compiled based on extensive experimentation, appears in Rostin et al. [51]. Some of the most important rules are:

1. A foreign key should have significant cardinality;
2. A foreign key should have good coverage of the primary key;
3. A foreign key should not be at the same time a primary key for too many other foreign keys;
4. The set of values of a foreign key should not be a subset of too many primary keys;
5. The average length of the values in foreign/primary key columns should be similar (mostly for strings);
6. The primary key should have only a small percentage of values outside the range of the foreign key;

7. The column names of foreign/primary keys should be similar.

Indeed, a lot of previous work, especially in the realm of schema matching, has used similar rules to find associations between columns [35].

It is important to note that counter-examples exist for any rule one tries to devise. One can easily come up with such examples for rules 1 and 2. A counter-example for rules 3 and 4 is when social security numbers or telephone numbers are used in a database as primary keys. Such keys are expected to appear in a large number of tables. Kang and Naughton [35] give several counter-examples for rule 7, i.e., columns with no meaningful association but very similar names.

In this work, we propose a novel method for measuring the likelihood that a pair of columns that satisfy inclusion is a useful fk/pk constraint. Our approach subsumes a variety of previous rules, and, as we show in Section 3.6, is both highly scalable and accurate.

Consider the set of values in a primary key P, ordered by the natural order in the underlying domain (i.e., numerical order for numeric attributes, lexicographic order for strings). We conjecture that in most cases the values in a foreign key column F form a (nearly) uniform random sample of the values in P. For example, consider columns Broker.BID and Customer Account.BID from Figure 3.1. The broker ids that appear in Customer Account are expected to be “sprinkled” uniformly throughout the ordered set of all broker ids. This is because we have no reason to expect a correlation between the semantics of the foreign key constraint (“this broker works with these customers”) and the mechanism through which the broker ids are generated. For example, ids may be consecutive numbers between 1 and 100 generated via auto-increment. By contrast, the subset of broker ids that appear in Customer Account may reflect, say, those brokers with great reputation. It is highly unlikely that a database instance is designed such that a foreign key is a biased sample of the respective primary key (e.g., a prefix or a suffix in the ranked order). Even if this is the case, for dynamic databases the distribution of fk/pk values is expected to change over time, eventually eliminating such bias. The closer a column F is to being a uniform random sample of a primary key P, the higher we consider the likelihood that (F, P) is a useful fk/pk constraint.

Randomness is a strong requirement that implies rules 1-6: If F is a uniform random sample of P, rule 2 (and by extension rule 1 relative to the cardinality of P) is satisfied. Similarly for rule 6. If the underlying distribution of column F

is the same as column P, and F is a random sample of P, then the probability that a substantial number of columns F' are random samples of F, without any real correlation between F and F', is very small (rule 3). Similarly, if F is a random sample of P, and F is a random sample of some other column P' with the same underlying distribution as P, then P and P' are clearly highly correlated. First, it is unlikely that a large number of such correlated columns P' exist (rule 4). Second, any such association (F, P') has high confidence if (F, P) has high confidence, so it is equally valid. Finally, if F is a random sample of P rule 5 is straightforwardly satisfied.

Nevertheless, one can come up with counter-examples for the randomness rule as well. Consider a data warehouse that contains a table P with all historical transactions, and a table F that references only the last month of transactions (for the purpose of efficiently answering queries on the latest data). If transaction identifiers are assigned using an auto-increment field, then the transaction id field in table F is a foreign key to table P and the transaction id values in F form a suffix of the ids in P. Note that this example also invalidates rule 6. Unless a foreign key constraint is specified in the schema, no formal method can decide *with 100% certainty* whether a column F is a foreign key with respect to primary key P. As mentioned above, useful fk/pk relationships are often data-dependent, and may not be specified in the schema. Hence, we cannot expect to find a solution with 100% precision/recall. However, as we show via extensive experiments over a large number of real databases, the randomness rule eliminates a very large number of false positives in practice.

In this chapter we show that *Randomness* efficiently discovers meaningful foreign keys, including multi-column foreign-keys (which have not been considered by previous work). Our experiments show that our approach has higher accuracy than previously proposed methods, scales to very large datasets, and does not require any prior knowledge of the data (in contrast with the method in [51]). Our contributions in this chapter are summarized as follows:

- We define a distance measure between distributions, which allows us to quantify *Randomness*. This leads to a novel foreign key discovery rule that prunes a large number of false positives.

- We design fast approximate algorithms for evaluating randomness over a large set of columns, using quantile summaries.

- We design an I/O efficient algorithm for discovering single and multi-column foreign keys, which requires only two linear scans of the data. It outputs a list of fk/pk pairs, in descending order of their randomness scores. The score reflects the likelihood that the pair is a useful foreign key constraint.

- We present a comprehensive experimental validation of our approach using a large number of real and synthetic datasets.

3.2 Preliminaries

We assume that the single and multi-column primary keys are known, either from schema specification or from a preprocessing phase. In the latter case, GORDIAN [54] can be used to compute them. We now formalize the problem of foreign key discovery as follows:

Definition 3.1 (Foreign Key Discovery)**.** *Let* $\mathbf{T}$ *be a collection of relational tables, possibly from multiple databases. Let* $\mathbf{C}$ *be the set of all columns in* $\mathbf{T}$*. Let* $\mathbf{P}_s$ *(*$\mathbf{P}_m$*) denote the set of single-column (multi-column) primary keys and* $\mathbf{P} = \mathbf{P}_s \cup \mathbf{P}_m$*.* Foreign Key Discovery *is the process of discovering the set of single-column and multi-column foreign keys with respect to* $\mathbf{P}$*.*

For the remainder of this chapter, we use F (P) to refer to both a single and multi-column foreign (primary) key. Abusing notation, F and P refer both to the names of the columns (or multi-columns) and to their respective set of distinct values (or tuples). In general, F and P are multi-sets. Let $|X|$ be the number of *distinct* values in multi-set X (or distinct tuples if X is multi-column). Table 3.1 summarizes the notations used in this chapter.

In order to cope with data inconsistencies, we relax the inclusion property that a foreign key must satisfy. More precisely, we require that

$$\sigma(F, P) = \frac{|F \cap P|}{|F|} \geq \theta,$$

where $\sigma(F, P)$ is the *inclusion coefficient* and θ is user-defined. In our experiments, we use $\theta = 0.9$, i.e., partial inclusion is satisfied if at least 90% of the values in F are also contained in P. We use the notation $F \subset_\theta P$ to denote that $\sigma(F, P) \geq \theta$.

Symbol	Description
$\mathbf{T}$	Set of tables
$\mathbf{C}$	Set of columns
$\mathbf{P}_s$	Single-column primary keys
$\mathbf{P}_m$	Multi-column primary keys
$\mathbf{F}_s$	Single-column candidate foreign keys
$\mathbf{F}_m$	Multi-column candidate foreign keys
B	Hash table for bottom-k sketches
Q	Hash table for quantile/distribution histograms
F	Single/multi-column candidate foreign key
P	Single/multi-column primary key
$\widehat{C}$	Bottom-k sketch of C
$\widehat{F}$	Bottom-k sketch of F
$\widehat{P}$	Bottom-k sketch of P
$\bar{P}$	Quantile histogram of P
$\bar{F}_P$	Distribution histogram of F with respect to P
$\sigma(F, P)$	Inclusion coefficient
θ	User-defined threshold for inclusion coefficient

Table 3.1: Notation used throughout Chapter 3.

Computing $\sigma(F, P)$ is very expensive, especially when considering the potentially very large number of multi-column candidate fk/pk pairs. Therefore, we estimate all inclusion coefficients by computing a bottom-k sketch [21] for each column . We briefly review bottom-k sketches as below.

Given a set F, a bottom-k sketch $\widehat{F}$ for F is computed as follows: Assign ranks to all values in F uniformly at random, and let $\widehat{F}$ be the set of k values with the smallest ranks. In practice, to compute the rank assignment we choose a hash function h, hash each value in F, and keep the k values corresponding to the smallest k hash values. If F is a set of tuples, rather than simple values, we first concatenate all values in a tuple using a predefined field separator and hash the resulting string as a whole. Figure 3.2 shows an example bottom-1 sketch for a set of tuples. Clearly, a bottom-k sketch can be computed in one pass over F.

Bottom-k sketches have been used to estimate various measures, such as the Jaccard coefficient $\rho(F, P) = \frac{|F \cap P|}{|F \cup P|}$ (see [21]) or the intersection size $|F \cap P|$ (see [11]). The estimators require that the same hash function h be used for computing both bottom-k sketches $\widehat{F}$ and $\widehat{P}$ (hence, the sketches are called *coordinated*).

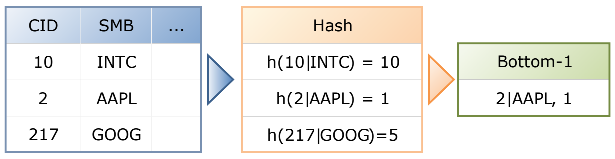


Figure 3.2: Constructing a Bottom-k sketch.

Section 3.4 provides details on how to efficiently compute bottom-k sketches for both single and multi-column candidate keys. We then use the SCS estimator from [21], which estimates the Jaccard coefficient $\rho(F, P) = \frac{|F \cap P|}{|F \cup P|}$. Since $\sigma(F, P) = \frac{\rho(F,P)}{\rho(F \cup P, F)}$, we estimate $\sigma(F, P)$ by dividing the estimators for the two Jaccards. In Section 3.6 we discuss two alternative estimators we used in our experiments. Each has a significant drawback. By contrast, this estimator proved highly accurate.

3.3 Randomness

In this section we assume that the inclusion coefficients between all pairs of single/multi-column pks and columns in $\mathbf{C}$ have been computed, and pairs that do not satisfy partial inclusion have been discarded. As mentioned in Section 3.1, we conjecture that randomness is a strong indicator of the quality of an fk/pk pair. Formally:

Definition 3.2 (Randomness test). *Given two sets of values (tuples) F and P, test the statistical hypothesis that the distinct values (tuples) in F have the same underlying distribution as the distinct values (tuples) in P.*

Figure 3.3 shows an example of a two-column primary key and two candidate foreign keys. Set F is a good fk, since it appears to be a random subset of values from the pk. Set F' is a contiguous subset of the pk and does not pass the randomness test.

Domain Order. The randomness test requires the existence of an underlying order over the domain of the primary and foreign keys. To see this, consider the example in Figure 3.4. If the values are sorted numerically, then the candidate column F is a prefix of the primary key. However, when the same values are sorted lexicographically, F falsely appears to be a random sample of the primary key. To handle this issue, we adopt the following nat-

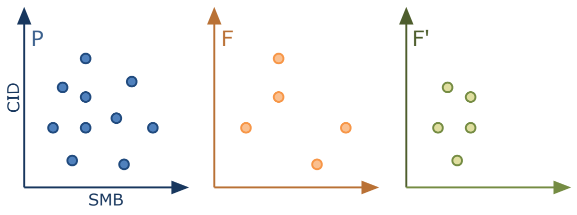


Figure 3.3: A good foreign key F is a set of random values from the primary key. Column F' fails the randomness test.

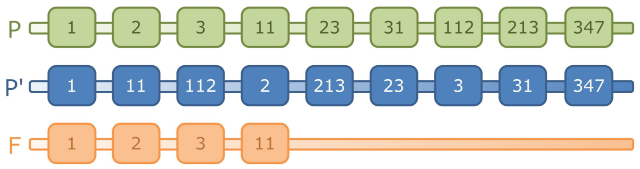


Figure 3.4: A column containing numeric values might falsely appear to be a random sample of a primary key based on lexicographic sorting of values.

ural convention: numeric values are sorted numerically, and strings are sorted lexicographically. The implicit assumption is that it is very rare that a column containing only numeric values is a foreign key for a primary key that contains strings (in which case it should have been sorted lexicographically, rather than numerically). When columns contain both numeric, alphanumeric, and string values, we use a combination sort (same as the Unix "sort -n" command). For multi-column keys we define an order along each dimension, as above.

Randomness measure. A standard, non-parametric statistical test for randomness is the Wilcoxon rank-sum test [53]. Assume that F, P are single-column candidate keys. Sort the values in the multi-set union $F \cup P$ and rank them. Since $F \subset_\theta P$, the majority of values in F appear in P, so there are duplicate values. Assign the mean rank for duplicate values (i.e., if a duplicate value is 3rd and 4th in the sorted order, it is assigned rank 3.5; see Figure 3.5). Finally, compute the sum of ranks of all values in F. This rank-sum is an indication of whether F and P are drawn from the same distribution. Intuitively, if the rank-sum is too small, then most values in F are contained in a prefix of P, and if the rank-sum is too large then most values in F are contained in a suffix of P.

The Wilcoxon test is straightforward for univariate distributions but does not generalize to multivariate distributions, so it cannot be used for multi-column keys. Attempting to apply the Wilcoxon test separately for each di-

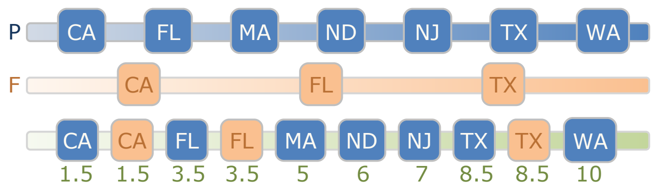


Figure 3.5: The Wilcoxon test: 1. Sort values in multi-set $F \cup P$; 2. Assign ranks; 3. Compute the rank-sum of values in F (13.5 in this example).

mension of a multi-column key results in false negatives. For example, consider the multi-column key F in Figure 3.3. Even though F appears to be a uniform random sample of P, the projection of F in either dimension is not a uniform sample due to the multiplicity of some of the values (two points project into the same value in both dimensions). An independent Wilcoxon test in either dimension would dismiss F.

We now propose a novel approach for deciding whether two multi-dimensional sets are drawn from the same distribution. Our method computes a value that reflects how close the distributions of the two sets are. We start by defining a probability distribution for each set, so that the total probability mass is 1 (this step is detailed later in the section). A standard distance measure between two probability distributions is the Earth Mover's Distance (EMD) [45]. Formally, Earth Mover's Distance is defined as follows:

Definition 3.3. *Given probability density functions (pdfs) C and C' on an underlying metric space, let a unit amount of work be equal to moving a unit of probability mass for a unit distance. Then, $EMD(C, C')$ is equal to the minimum amount of work needed to convert pdf C into pdf C'.*

The smaller the value $EMD(F, P)$ is, the closer the distributions of F and P are. The output of our algorithm is the list of (F, P) pairs, in increasing order of their (normalized) EMD values.

Intuitively, EMD measures the amount of work needed to convert the set of values of the foreign key into the set of values of the primary key. If we regard each distribution as piles of dirt spread over some space, EMD is the least amount of effort needed to convert the first set of piles into the second. The effort is the amount of dirt that needs to be moved times the distance it has to travel. Figure 3.6 illustrates the computation of EMD for pairs (F, P) and (F', P) from Figure 3.3. In this example, all points in a set have equal

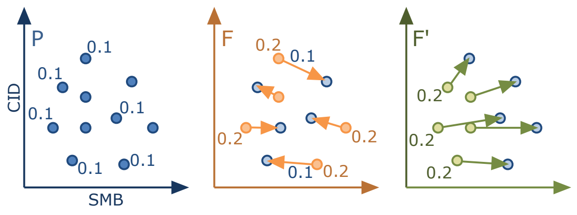


Figure 3.6: EMD quantifies the amount of work required to convert one set of values into another.

probability and the sum in each set is equal to 1. To convert F into P, a probability mass of 0.1 needs to be moved from each point $p \in F$ to the nearest point $np(p) \in P \backslash F$. Similarly for F' and P. Since the points of F are uniformly distributed over P the average distance between p and $np(p)$ is smaller than the average distance for F'. Hence, the amount of work needed to convert distribution F to P is smaller than the one to convert F' to P.

While the definition of EMD applies to single and multi-dimensional sets, it has a crucial restriction: unlike the Wilcoxon test, EMD requires a metric distance between the values of the two distributions. A metric distance can be used only when both columns F and P contain numeric values, but not when they contain strings. Even for numeric values, using the underlying distance is undesirable because we need to be able to compare EMD values between different candidate pairs for sorting pairs according to confidence. However, given distinct F, F', P, P', if F and P have larger ranges of values than F' and P', then $EMD(F, P)$ will generally tend to be larger than $EMD(F', P')$, even if F is a "more random" subset of P than F' is for P'. Therefore, a uniform way of defining a distance function for numeric and string columns is needed, which is independent of the range of values in any column.

We propose using the distance between the ranks of the values in the pk column. For single-column F and P, rank all values in P in the underlying ordered space, then define the rank distance between two values in F or P to be the (absolute) difference between their ranks in P. For multi-columns F and P, define the rank distance to be the sum of single-dimensional rank distances (i.e., the Manhattan distance). However, the rank distance will still introduce bias when comparing $EMD(F, P)$ and $EMD(F', P')$ if the number of values in $|P|$ is much larger than the number of values in $|P'|$. Therefore, the rank distance is normalized by the number of values, in effect replacing ranks by quantiles:

Definition 3.4 (Quantile Distance)**.** *Given a multi-column set X consisting of n columns, a total order in each column X_i, a function $q_i(x)$ that returns the quantile order of value x in column X_i, and two tuples $v, w \in X$, the quantile distance is*

$$d(v, w) = \sum_{1 \le i \le n} |q_i(v) - q_i(w)|.$$

Notice that the quantile distance is independent of the type of values in X as long as a total ordering of the values *in each dimension* is defined. We refer to the EMD measure using the quantile distance as *Quantile-EMD*. A final normalization is needed to compare (F, P) and (F', P') when they have different dimensionality. Let $EMD_n(F, P) = EMD(F, P)/n$, where n is the dimensionality of F and P.

Computing Quantile-EMD. We now consider the problem of efficiently approximating $EMD(F, P)$ for all pairs of candidate keys (F, P). The first step is to define a probability distribution for F and P. The easiest choice is to let each value in F have a probability mass of $1/|F|$, and each value in P have a probability of $1/|P|$. Computing EMD is equivalent to the well-known transportation problem and can be solved by the Hungarian algorithm [23]. However, the Hungarian algorithm has cubic complexity and is very inefficient over large F and P. For our purposes, it is sufficient to compute EMD on coarser probability distributions. More precisely, we use a quantile histogram to define the probability distribution in the primary key, since quantiles best approximate the original distribution w.r.t. the quantile distance. The probability distribution in the candidate foreign key is then defined with respect to the quantiles of the primary key.

For every single/multi-column key $P \in \mathbf{P}$ construct a *quantile histogram* based on the ℓ-quantiles of P (for some constant ℓ). In one dimension, the histogram is equi-depth. In multiple dimensions, compute quantiles separately on each dimension (over the *distinct* values in that dimension) and construct a grid based on the quantiles in each dimension. An example 2-dimensional 4-quantile histogram is shown in Figure 3.7. Notice that in this particular example there exists a three point tie in each dimension. After projecting the points in either dimension there are only 8 distinct values left. Hence, the 1st 4-quantile is the point with rank $8 \cdot 1/4$, the 2nd is the one with rank $8 \cdot 2/4$,

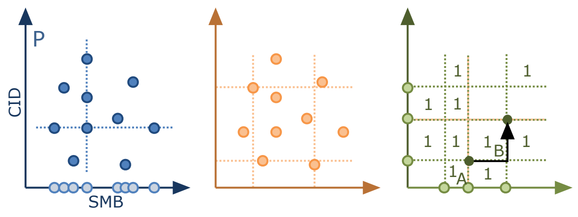


Figure 3.7: Constructing a 2-dimensional 4-quantile histogram for primary key P.

etc. The probability distribution of P based on the corresponding histogram is defined as:

Definition 3.5 (Quantile Histogram)**.** *Given a multi-column primary key P consisting of n columns, let $Q_i = \{q_1^i, \ldots, q_{\ell_i}^i\}$ be the ℓ_i-quantiles of P in column i (different columns may have different number of quantiles). Let $G_P = Q_1 \times \ldots \times Q_n$ be the corresponding n-dimensional quantile grid. The quantile histogram $\bar{P}$ is defined as the number of values of P within each grid cell of G_P. The total number of grid cells is $|G_P| = \ell_1 \times \ldots \times \ell_n$. The probability distribution over P is defined as the normalized $\bar{P}$; i.e., the count in each cell is divided by $|P|$.*

For a candidate multi-column F, the probability distribution histogram based on the quantile grid G_P of P is defined as:

Definition 3.6 (Distribution Histogram)**.** *Given a candidate pair (F, P), the distribution histogram $\bar{F}_P$ of F with respect to P is defined as the number of distinct values of F within each grid cell of the quantile grid G_P. The probability distribution over F is defined as the normalized $\bar{F}_P$; i.e., the count in each cell is divided by $|F|$.*

We now describe how to approximate $EMD(F, P)$ using the quantile histograms. Assume that the probability mass of a grid cell is concentrated in its upper right corner. Therefore, the distance between two grid cells is defined as the quantile distance between the upper right corners of the cells. For example, in Figure 3.7, the distance between grid cells A and B is $(3/4 - 2/4) + (2/4 - 1/4) = 0.5$. As before, the Hungarian algorithm is used to compute the EMD between the two distributions. The input size is now $|G_P| = \ell^n \ll |P|$ (usually $1 \leq n \leq 4$ and ℓ is small). Once the normalized histograms are computed, the method requires no additional access to the raw data. Note that the value ℓ need not be the same for all primary keys. Since Quantile-EMD uses the quantile distance we can compare $EMD(F, P)$ and $EMD(F', P')$ even if the quantile histograms were computed for different values of ℓ. This is important, since some primary keys may have only a few values. On the other hand, a larger ℓ for larger primary keys will improve accuracy.

Now we can bound the approximation error for the Quantile-EMD in the grid space G_P versus the initial space P.

Lemma 3.1. *Let n be the space dimensionality, $1/\ell$ be the side length of the cells in G_P (in every dimension without loss of generality), and $EMD_{n,P}$, EMD_{n,G_P} be the normalized EMD in the primary space P and reduced space G_P respectively. Then*

$$|EMD_{n,P} - EMD_{n,G_P}| \leq 2/\ell.$$

Proof. Let $\mathrm{EMD}_P{=}n{\cdot}\mathrm{EMD}_{n,P}$ and $\mathrm{EMD}_{G_P}{=}n{\cdot}\mathrm{EMD}_{n,G_P}$ be the unnormalized EMD values in the primary space P and the reduced space G_P respectively. Consider a movement of mass m that EMD_P executes in the primary space P, from a point p to a point q. Its cost is $m \cdot d(p, q)$. Let a and b be the upper right corners of the cells that contain p, respectively q. Then we can define a valid movement of mass m in the space G_P, between a and b. The cost of this mass movement is $m \cdot d(a, b) \leq m(d(a, p){+}d(p, q){+}d(q, b)){\leq} md(p, q) + m\frac{2n}{\ell}$. Making this transformation for all mass movements in EMD_P, we obtain a valid mass movement in G_P, of cost at most $\mathrm{EMD}_P + 2\frac{n}{\ell}\sum m \leq \mathrm{EMD}_P + 2\frac{n}{\ell}$ (the sum is over all the mass moved in EMD_P). Since EMD_{G_P} is the *minimum* cost movement in G_P, we deduce $\mathrm{EMD}_{G_P} \leq \mathrm{EMD}_P + 2n/\ell$. A similar argument holds for the other inequality, by transforming mass movements from EMD_{G_P} into valid mass movements in P. We deduce that $|\mathrm{EMD}_P - \mathrm{EMD}_{G_P}| \leq \frac{2n}{\ell}$. □

3.4 Overall Algorithm

Throughout this section, we use the notations from Section 3.2 (see also Table 3.1 for a notation summary). To discover foreign keys, we first compute inclusion between all pairs of primary keys and columns in **C** and then evaluate randomness only on the pairs satisfying inclusion. We accomplish this by computing bottom-k sketches and quantile histograms with two passes over the data. A pseudocode of the algorithm described below appears in Algorithm 3.1.

For single-column candidate foreign keys and single/multi-column primary keys, the bottom-k sketches can be computed in one linear scan of the database. Multi-column candidate foreign keys are challenging due to their potentially large number. However, if P is a primary key consisting of columns $(C_1, \ldots, C_n)$ and F is a candidate foreign key consisting of columns $(C'_1, \ldots, C'_n)$, then $\sigma(F, P) \leq min_{i=1}^{n} \sigma(C'_i, C_i)$. Hence, it is sufficient to consider only candidates F such that $\sigma(C'_i, C_i) \geq \theta$, for all i, where all C'_is belong to the same table. We expect only a small number of pairs (F, P) to have these properties (and confirm this experimentally). For such pairs (F, P), we compute the bottom-k sketch of F and estimate $\sigma(F, P)$, with a second pass over only the relevant columns in F (recall that the sketch of P has been computed during the first pass).

Quantile histograms for a single-column primary key P can be *computed exactly* in one linear scan if there exists an index on P – as is usually the case for pks – by reading P in sorted order and computing the quantiles incrementally (this requires knowledge of $|P|$ which can be found from table statistics). If an index does not exist we can *approximate* the quantiles in linear time using quantile summaries [30]. The distribution histograms of single-column candidate fks can be trivially computed in linear time after the quantile histograms of all pks have been computed and stored in memory; for an fk F we simultaneously compute all histograms w.r.t. all pks P for which (F, P) passes inclusion. In our experiments, the average number of such pairs, for a fixed F, was less than 10.

For a multi-column primary key P, the quantile histogram requires two passes over the data. In the first pass, we compute the quantile grid for each column $C \in P$ (either using an index or a quantile summary) and construct a multi-dimensional quantile grid. A subtle point here is that a column C of a multi-column pk might not be a pk itself; estimating quantiles on the distinct values in C requires using duplicate insensitive quantile summaries [22]. In the second pass, we scan P and populate the quantile grid. We also scan each multi-column fk F and simultaneously compute all relevant distribution histograms. In our experiments, the average number of such pairs, for a fixed F, was less than 5.

We now summarize each of the two linear scans:

Algorithm 3.1: Discover Foreign Keys $(\mathbf{C}, \mathbf{P}_s, \mathbf{P}_m, \theta)$

```
   // Phase 1.
1  $\mathbf{F}_s \leftarrow \emptyset, \mathbf{F}_m \leftarrow \emptyset, B \leftarrow \emptyset, Q \leftarrow \emptyset, S \leftarrow \emptyset$
2  foreach $C \in \mathbf{C}$ do
3      $B[C] \leftarrow \widehat{C}$
4  foreach $P = \{C_1, \ldots, C_n\} \in (\mathbf{P}_s \cup \mathbf{P}_m)$ do
5      for $p \leftarrow 1$ to $n$ do
6          foreach $C_f \in \mathbf{C}$ do
7              if $\sigma(\widehat{C}_f, \widehat{C}_p) \geq \theta$                /* $\widehat{C}_f, \widehat{C}_p \in B$ */
8              then
9                  if $n = 1$ then
10                     $\mathbf{F}_s \leftarrow (C_p, C_f)$
11                 if $n > 1$ then
12                     $S[P, C_p] \leftarrow C_f$
13     if $n > 1$ then
14         $B[P] \leftarrow \widehat{P}$
15     $Q[P] \leftarrow \bar{P}$                /* For $n=1$, $\widehat{P}$ already in $B$ */
   // Phase 2.
16 foreach $P = \{C_1, \ldots, C_n\} \in \mathbf{P}_m$ do
17     foreach $T \in \mathbf{T}$ do
18         $\mathbf{F}_m \leftarrow (\{\{C'_1, \ldots, C'_n\} \mid C'_i \in S[P, C_i] \cap T\}, P)$
19     foreach $F = (\{C'_1, \ldots, C'_n\}, P) \in \mathbf{F}_m$ do
20         Build $\widehat{F}$
21         if $\sigma(\widehat{F}, \widehat{P}) \geq \theta$                /* $\widehat{P} \in B$ */
22         then
23             $Q[P] \leftarrow \bar{P}$
24             $Q[F] \leftarrow \bar{F}_P$
25         else
26             Remove $(F, P)$ from $\mathbf{F}_m$
27 foreach $(F, P) \in \mathbf{F}_s$ do
28     $Q[F] \leftarrow \bar{F}_P$
29 foreach $(F, P) \in (\mathbf{F}_s \cup \mathbf{F}_m)$ do
30     Compute $EMD_n(F, P)$                /* Using $Q$ */
31 Output $\mathbf{F} = \mathbf{F}_s \cup \mathbf{F}_m$ in increasing order of $EMD_n$
```

Phase 1. Read all columns in table-wise order (i.e., row by row) and build bottom-k sketches for all columns in $\mathbf{C}$, as well as for all multi-column primary keys in $\mathbf{P}_m$. Also build quantile grids for all single/multi-column primary keys. All structures are stored in two hash tables B (for bottom-k sketches) and Q (for quantile/distribution histograms), using the name of the column(s) as the hash key. Evaluate all (single-column) inclusions between $F \in \mathbf{C}$ and $P \in \mathbf{P}_s$ and store candidate pairs in $\mathbf{F}_s$. Finally, evaluate (single-column) inclusions between $C \in \mathbf{C}$ and $C_i \in P, P \in \mathbf{P}_m$ and store candidates in $S(C_i)$.

Phase 2. For each multi-column pk $P = (C_1, \ldots, C_n)$, consider the n sets $S(C_i) = \{C \in \mathbf{C} \mid \sigma(C, C_i) \geq \theta, 1 \leq i \leq n\}$ computed in Phase 1. Then $F = (C'_1, \ldots, C'_n) \in S(C_1) \times \ldots \times S(C_n)$ is a candidate fk for P if there exists a table T s.t. $\forall 1 \leq i \leq n : C'_i \in T$. Compute, for each multi-column pk P the set of its candidate foreign keys and insert pairs (F, P) in $\mathbf{F}_m$. This requires access only to the sets $S(C_1), \ldots, S(C_n)$, which are stored in memory.

Next, for each $(F, P) \in \mathbf{F}_s$, scan each single-column F and compute its distribution histograms w.r.t. all relevant primary keys P. Compute $EMD_1(F, P)$ and store it in memory. For each multi-column candidate $(F, P) \in \mathbf{F}_m$ scan F and compute its multi-column bottom-k sketch as well as its distribution histograms w.r.t. all P. For each such P, verify whether (F, P) satisfies inclusion (recall that the multi-column bottom-k sketch of P was computed in Phase 1). If (F, P) does not pass the test, discard the distribution histogram $\bar{F}_P$ and remove (F, P) from $\mathbf{F}_m$. Finally, compute $EMD_n(F, P)$ for all $(F, P) \in \mathbf{F}_m$ and return $\mathbf{F} = \mathbf{F}_s \cup \mathbf{F}_m$, sorted in increasing order of EMD_n values.

3.5 Schema and Data Updates

Our methods can easily handle insertions and deletions of new tables and columns given the existing bottom-k sketches and quantile/distribution histograms. Let the new set of columns be $\mathbf{C}'$. First, identify new primary keys and insert them in $\mathbf{P}_s, \mathbf{P}_m$. Then, re-run Algorithm 3.1 on $\mathbf{C}', \mathbf{P}_s, \mathbf{P}_m, \theta$, building only the new bottom-k and quantile/distribution histograms, as necessary.

Handling data insertions and deletions on existing columns is a little harder. Existing bottom-k sketches can easily be updated under insertions only. The new values are simply hashed and inserted in the corresponding sketches if necessary. However, deletions are not straightforward: if a deleted value was part of the bottom-k sketch, a rescan of the corresponding column is needed in order to identify the new k-th minimum hash value. One way to handle deletions without rescanning the data is to maintain larger bottom-k sketches (e.g., twice as large as needed). That way, we only rescan the data infrequently. Since in practice we expect a balanced insertions and deletions workload, this simple strategy is likely to obviate the need of a rescan in most settings.

Updating the quantile/distribution histograms is generally hard, both under insertions and deletions. A small number of insertions or deletions can be accommodated by identifying the histogram cells that contain the respective tuples, and incrementing or decrementing their counters. However, if a large amount of data is inserted or deleted, the distribution of the underlying columns is likely to change. As a result, the quantile grids on each dimension also change. This requires rescaning the data in order to compute a new quantile grid and a new histogram. A simple way of reducing the cost of updates is to use the existing quantile grid for a batch of updates and rebuild it only after a certain number of updates. Depending on the application and the underlying data we can also use well known techniques to detect a change in the distribution and trigger a rebuild [36].

3.6 Experimental Evaluation

We evaluate our algorithm for discovering fk/pk constraints on two benchmark synthetic databases (TPC-E and TPC-H), as well as on two real datasets: a Wikipedia (WP) snapshot from March 2008 and an IMDB snapshot from January 2010. We implemented our algorithms in C++, and performed the experiments on an Intel Core2 Duo 2.33 GHz CPU with 4GB RAM running MySQL. We use three standard accuracy measures to evaluate our method: *precision*, *recall* and *F-measure* (the harmonic mean of precision and recall). Each measure is applied to two sets of fk/pk constraints: the “golden standard” set specified in the schema, and the top-X% constraints reported by our algorithm (for various X). We start by evaluating the accuracy of EMD computation, then evaluate the overall algorithm for both accuracy and scalability. In addition, we discuss how the results change if we also take into account the similarity of column names. Finally, we compare our results with the machine learning approach of Rostin et al. [51], which uses the 7 rules discussed in Section 3.1. For completeness, we also include experiments on bottom-k sketches, which show that the partial inclusion estimator is highly accurate and does not influence the overall results.

3.6.1 Dataset Descriptions

The datasets can be downloaded from the following sites: TPC-H from `http://www.tpc.org/tpch`, TPC-E from `http://www.tpc.org/tpce`, WP from `http://www.archive.org/details/enwiki-20080312`, IMDB from `http://www.imdb.com/interfaces`.

When generating instances for the synthetic datasets, we use the following parameter settings: For TPC-H we use scale factor 1. For TPC-E we use 1000 customers, 20 trading days, and scale factor 1000. The characteristics of all datasets are given in Table 3.2, where $|\mathbf{T}|$ is the number of non-empty tables, $|\overline{\mathbf{C}_T}|$ and max $|\mathbf{C}_T|$ are the average and maximum number of columns per table, and $|\overline{\mathbf{R}_T}|$ and max $|\mathbf{R}_T|$ are the average and maximum number of rows per table.

	$\lvert\mathbf{T}\rvert$	$\lvert\overline{\mathbf{C}_T}\rvert$	max $\lvert\mathbf{C}_T\rvert$	$\lvert\overline{\mathbf{R}_T}\rvert$	max $\lvert\mathbf{R}_T\rvert$
TPC-H	8	8	16	1,082,504	6,000,003
TPC-E	32	6	24	171,127	4,469,625
WP	15	6	16	24,356,005	227,867,141
IMDB	9	2	2	1,136,607	5,107,802

Table 3.2: Datasets characteristics.

All these datasets come with a schema specification. Table 3.3 summarizes the single/multi-column foreign/primary keys explicitly stated in each schema. Notice that, e.g., TPC-E specifies nine 2-column primary keys but only one 2-column foreign key.

		TPC-H		TPC-E		WP		IMDB	
		PK	FK	PK	FK	PK	FK	PK	FK
SC		6	9	20	44	5	10	5	8
MC	2	2	1	9	1	7	–	4	–
	3	–	–	2	–	3	–	–	–
	4	–	–	1	–	–	–	–	–
Total		8	10	32	45	15	10	9	8

Table 3.3: Foreign/primary keys according to schema specifications.

ℓ		4	16	64	256	1024	256(A)
Diff		0.06	0.009	0.002	$3\ 10^{-4}$	$4\ 10^{-5}$	$4\ 10^{-4}$
Top-25%	R	1	1	1	1	1	1
	P	0.9	0.9	0.9	0.9	0.9	0.9
Top-20%	R	0.78	0.78	0.89	0.89	0.89	0.78
	P	0.88	0.88	1	1	1	0.88
Top-15%	R	0.56	0.67	0.67	0.67	0.67	0.56
	P	0.83	1	1	1	1	0.83

Table 3.4: EMD accuracy for different quantile grid sizes; Diff=$EMD_{n,G_\ell} - EMD_{n,G_{2048}}$.

3.6.2 EMD Computation

By Lemma 3.1, the error in the computation of $EMD(F, P)$ is bounded by $2/\ell$, where ℓ is the number of quantiles in each dimension. In practice, this error is negligible even for small ℓ. Table 3.4 shows the average difference between the EMD computed using $\ell \in \{4, \ldots, 1024\}$ quantiles and 2048 quantiles. The averages are over all columns in the TPC-H database that pass inclusion. The reason we do not compute the differences to the actual EMD value (i.e., using all quantiles) is that EMD has cubic complexity, which is very expensive to compute when the primary key has $\Omega(10^4)$ values. Clearly, the EMD values converge very quickly as ℓ increases. Therefore, small values of ℓ are sufficient.

The last column of Table 3.4 shows that, when using 256 *approximate quantiles*, the difference is only 10^{-4} bigger than for 256 exact quantiles. This is not surprising, since the quantile histograms of foreign keys should remain roughly the same over small shifts in the quantile grids. The main difference between approximate versus exact quantiles is computation time. To our surprise, computing exact quantiles was about 20% faster! The reason is that the vast majority of primary keys are indexed, so using an ORDER BY SQL query had the complexity of a linear scan. By contrast, computing approximate quantiles incurred the overhead of maintaining the quantile summaries [30]. Henceforth, all reported results are for exact quantiles.

Since normalized EMD values determine the order in which we report fk/pk pairs, we also measured the effect of quantile grid sizes on the precision and recall of the final results. Smaller grid sizes, as well as approximate quantiles, do impact both precision (P) and recall (R) for the top-15% and top-20% reported pairs (see last 4 rows in Table 3.4). However, they have no effect for the top-

Dataset	TPC-H		TPC-E		WP		IMDB	
SC-FK	9		44		10		8	
MC-FK	1		1		0		0	
$\theta =$	0.9	1	0.9	1	0.9	1	0.9	1
Cand. SC	38	34	304	214	12	8	24	24
Cand. MC	1	1	4	3	0	0	0	0

Table 3.5: Number of candidate pairs that satisfy inclusion; SC=single-column, MC=multi-column.

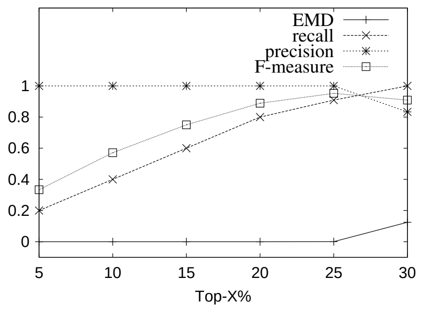
25% pairs. The reason is that EMD values are very close to each other within the top-25% pairs, so even small changes impact the order of results. However, pairs below top-25% have EMD values at least one order of magnitude larger. As we discuss below, we use precisely this jump in EMD values to determine the best set of constraints to present to a user. For TPC-H, the best set is the top-25%, which remains unchanged for all grid sizes.

3.6.3 Overall Algorithm

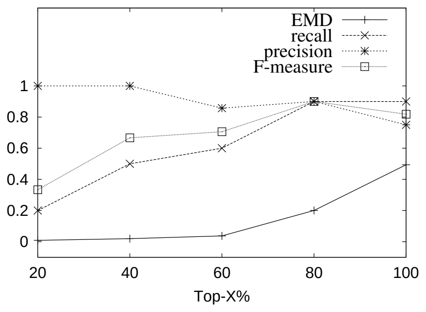
For all experiments in this section, we use $\ell = 256$ quantiles for single-column primary keys, and $\ell = 16$ quantiles per dimension for multi-column primary keys. Table 3.5 shows the number of candidate fk/pk pairs that pass the inclusion test for each dataset. This illustrates the large number of false positives our algorithm must eliminate. For example, 217 pairs pass inclusion for TPC-E (214 single-column pairs and 3 multi-column pairs). Of these, only 45 are specified in the schema.

To evaluate the utility of our method, we measure the precision, recall and F-measure after selecting the top-X% results as the answer set, and comparing them with the golden standard specified in the schema. We report the results in two groups, for reasons that we explain later in this section.

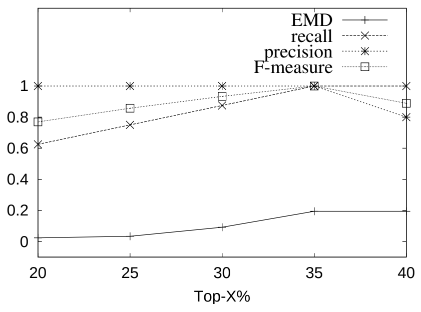
TPC-H, WP and IMDB: Figure 3.8 shows the results for these three databases. A larger X (i.e., a bigger answer set) implies a larger number of false positives, hence lower precision. On the other hand, a smaller X has more false negatives (i.e., undiscovered fk/pk constraints), and thus worse recall. There is a sweet spot, which depends on each dataset, that balances precision and recall. That spot also corresponds to a big jump in the respective EMD values; we illustrate this by plotting the EMD values on the same graph. For TPC-H



(a) TPC-H.



(b) Wikipedia.



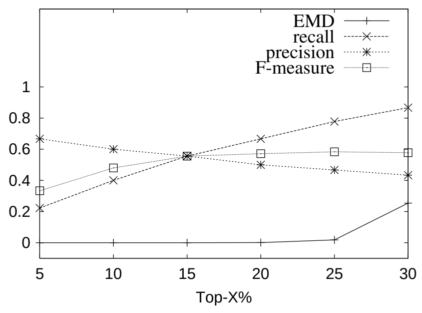
(c) IMDB.

Figure 3.8: Utility measures on TPC-H, Wikipedia and IMDB.

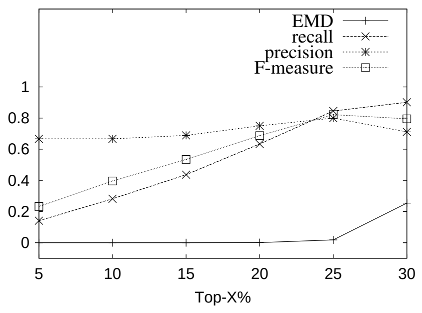
the sweet spot occurs at $X = 25$, for IMDB at $X = 35$, and for WP at $X = 80$. Thus, by examining the significant jumps in EMD values, the algorithm automatically proposes one or more answer sets deemed "relevant" (note that for WP, an EMD jump also occurs for $X = 60$). An answer set can then be verified either experimentally (e.g., by running queries and testing if the results are meaningful), or by a domain expert.

For all three datasets in Figure 3.8, we achieve F-measure above 0.8 at their respective sweet spot X. For TPC-H, the answer set (top-25%) has only one false positive: the pair of columns PartSupp.PS_AVAILQTY and Supplier.S_SUPPKEY, the first containing values from 1 to 9999 and second from 1 to 10000. Clearly, only a supervised algorithm would be able to recognize this false positive. For IMDB, the answer set is exactly the golden standard.

For WP, the loss in recall originates from an unlikely source. The schema specifies that ImageLinks.il_to is a foreign key to the primary key Image.img_name. However, its inclusion coefficient is only 51%. Since we set the inclusion threshold at 90%, this results in a false negative. (We note that three other specified fk/pk pairs have inclusion coefficient below 1, but higher than 0.9). Such data



(a) TPC-E.



(b) TPC-E extended.

Figure 3.9: Utility measures on TPC-E using the golden standard and extended constraints.

inconsistencies verify the intuition that real data does not always follow the ideal rules from database theory.

TPC-E: For this database, we report two sets of experiments. In the first set (Figure 3.9(a)) we measure precision/recall with respect to the golden s-tandard specified in the schema. Notice that the F-measure is below 0.6 across the board. A careful analysis of the data reveals that the reported accuracy is misleading: many false positives occur either because of *symmetry* or *transitivity*. Symmetry refers to the case when a pair of primary key columns (A, B) is specified in the schema as fk/pk in one direction, but not in the other. However, (B, A) is clearly a valid constraint in this case. Transitivity occurs when, for three columns A, B and C, with B and C being primary keys in their respective tables, the constraints (A, B) and (B, C) are specified in the schema, while (A, C) is not (although it is clearly valid). By applying symmetry and transitivity rules, we extend the set of valid constraints against which we test our results. Reporting precision/recall with respect to this augmented set of constraints improves results significantly; see Figure 3.9(b).

Nevertheless, none of the measures reaches 1. We attribute this to the data generation process itself. A total of 15 false positive pairs (some pairs are counted in both directions) are between only eight columns. These columns belong to seven different tables and contain exactly the same number of rows and number of distinct values: the numbers 1 to 5000. Clearly, only a domain expert can label them as false positives (in our case, we used the extensive TPC-E documentation). The algorithm also fails to discover 8 out of 45 true constraints and 28 implied constraints; the pairs are shown in Table 3.6. Five of

Foreign Key		Primary Key	
Column	Values	Column	Values
Exchange.ex_ad_id	1-4	*Address.ad_id*	1-7504
Company.co_ad_id	5-2504	*Address.ad_id*	1-7504
Customer.c_ad_id	2505-7504	*Address.ad_id*	1-7504
Trade.t_st_id	*B*	*Status type.st_id*	A,. . .,E
Broker.b_st_id	*A*	*Status type.st_id*	A,. . .,E
Company.co_st_id	*A*	*Status type.st_id*	A,. . .,E
Customer.c_st_id	*A*	*Status type.st_id*	A,. . .,E
Security.s_st_id	*A*	*Status type.st_id*	A,. . .,E

Table 3.6: False negatives in TPC-E (A=Active, B=Completed, C=Canceled, D=Pending, E=Submitted).

these foreign keys contain only one distinct value (either the status 'Completed' or 'Active'). Clearly, the generator assigns a default value for this column for every row in the table, since not all trades in Trade can be completed, while all trades in Broker are active. One column contains address identifiers 1 to 4 even though the corresponding primary key contains 7504 distinct addresses. Finally, the other two address columns are (almost) a prefix and a suffix of the primary key and constitute a counter-example for the randomness rule.

3.6.4 Scalability

We tested the scalability of our method on TPC-H, for which it is easy to generate instances of progressively larger sizes. We used five instances with sizes 1MB, 10MB, 100MB, 1GB and 10GB. The running times for each of the two phases, as well as the total time are shown in Figure 3.10. For readability, we use a logarithmic scale on both axes. As expected, each phase takes linear time. The second phase is faster because we only have to scan the columns that satisfy inclusion (while in the first case, we scan all columns). For the 10GB instance, the total running time is less than 2.5 hours, making our method applicable to enterprise-scale datasets.

3.6.5 Column Names

So far our foreign key discovery process has been a data-driven approach. However, it is easy to enhance it by considering the orthogonal approach of looking

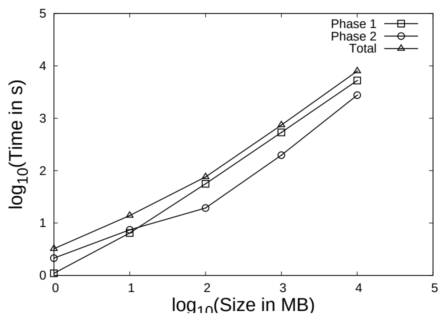


Figure 3.10: Scalability results.

at the column names (rule 7 in Section 3.1). As shown in [35], comparing column names is not necessarily straightforward, and can lead to false conclusions. For example, in TPC-E columns that form valid fk/pk constraints have very different names, because they contain an abbreviation of the table name as a prefix (e.g., columns Trade History.th_t_id and Trade.t_t_id are an fk/pk pair; the prefixes 'th' and 't' in the column names stand for Trade History and Trade respectively). Fortunately, TPC-E has extensive documentation that explains the naming conventions, so we can delete these prefixes and compare the remaining strings. The resulting names are identical only if the pair is a valid constraint.

We are not aware of any method for automatically determining which string similarity measure to use for any given schema. In Table 3.7, we report our results using string identity (for TPC-E, we apply this to column names after deleting their table prefixes). The results are generated as follows: First, we compute for each database the most relevant answer set, i.e., the top-X% for the best value X. We then delete all pairs from these sets whose column names are not identical, and compute the precision/recall on the resulting answer set. For TPC-E, we also report results using the extended set of valid constraints. For WP there is no single pair with identical column names, hence we exclude it from this experiment.

3.6.6 Comparison With Alternatives

The algorithm of Rostin et al. [51] uses a learning phase to train four different classifiers that are then used to discover single-column keys only. Each classifier uses a training set consisting of known fk/pk pairs from four out of five different

	Precision	Recall	F-measure
TPC-H	1	1	1
TPC-E	0.57	0.82	0.67
TPC-E Ext.	1	0.89	0.94
IMDB	1	1	1

Table 3.7: Results after eliminating non-matching column names.

databases. The goal is to learn the relative importance of rules 1-7 stated in Section 3.1, then apply them to the fifth database. No classifier was consistently the best across all datasets.

We compare our results over TPC-H using the results already reported in [51]. As reported in that paper, the best classifier (J48) for the TPC-H dataset results in F-measure equal to 0.95, with the average value over all classifiers being 0.915. The success of J48 for TPC-H can be largely attributed to the use of rule 7 (matching column names), making TPC-H an easy target. Our method achieves an F-measure of 1 for TPC-H when using column names (even without column names, F-measure is 0.95).

3.6.7 Inclusion Estimators

We considered two alternative estimators for the inclusion coefficient $\sigma(F, P)$:

1. The estimator proposed in [15], which is unbiased. However, it is defined over sketches whose sizes are a user-defined fraction of the size of the original column. This is generally too large for practical purposes (e.g., the Wikipedia database has size $O(10^9)$, so the size of 1%-sketches is $O(10^7)$). Note that the size of sketches impacts not just storage requirements, but more importantly, the running time for computing all inclusion coefficients.

2. Divide the estimated value of $|F \cap P|$ by the estimated value of $|F|$. Both estimated values are computed using the estimators proposed in [11]. The advantage is that these estimators work over bottom-k sketches which have constant size. Figure 3.11 shows an experimental comparison (over TPC-E) of this estimator, denoted Estimator2, to the one described in Section 3.2, which we denote as Estimator1. Clearly, Estimator1 is significantly more accurate, and it also uses bottom-k sketches. Therefore,

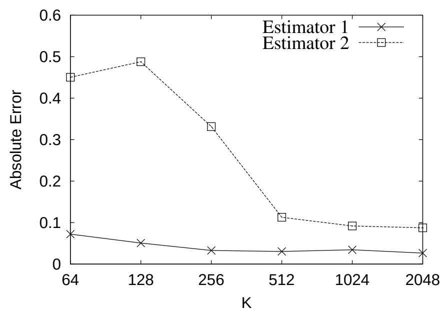


Figure 3.11: Accuracy of bottom-k estimators for the inclusion coefficient, as a function of k.

all our experiments use Estimator1 for the inclusion coefficient. We set $k = 256$.

3.7 Summary

In this chapter, we introduced the notion of *Randomness* and showed that it can be used effectively to reduce a large number of false positive pairs produced by partial inclusion. We also provided an efficient approximation algorithm for evaluating randomness between pairs of multi-column candidate keys. We presented a combined algorithm that can discover good single/multi-column foreign keys with only two linear scans over the data. Finally, we performed a comprehensive experimental evaluation showing the efficacy of our techniques.

This work has been published as a full research paper in *the* 36^{th} *International Conference on Very Large Data Bases* (VLDB) 2010 [59].

CHAPTER 4

Attribute Discovery

In this chapter we design algorithms for clustering relational columns into *attributes*, i.e., for identifying strong relationships between columns based on the common properties and characteristics of the values they contain. For example, identifying whether a certain set of columns refers to *telephone* numbers versus *social security* numbers, or names of *customers* versus names of *nations*. Traditional relational database schema languages use very limited primitive data types and simple foreign key constraints to express relationships between columns. Object oriented schema languages allow the definition of custom data types; still, certain relationships between columns might be unknown at design time or they might appear only in a particular database instance. Nevertheless, these relationships are an invaluable tool for schema matching, and generally for better understanding and working with the data. Here, we introduce data oriented solutions (we do not consider solutions that assume the existence of any external knowledge) that use statistical measures to identify strong relationships between the values of a set of columns. Interpreting the database as a graph where nodes correspond to database columns and edges correspond to column relationships, we decompose the graph into connected components and cluster sets of columns into attributes. To test the quality of our solution, we also provide a comprehensive experimental evaluation using real and synthetic datasets.

4.1 Introduction

Relational databases are described using a strict formal language in the form of a relational schema. A relational schema specifies various properties of tables and columns within tables, the most important of which is the type of data contained in each column. There is a well defined set of possible primitive data types, ranging from numerical values and strings, to sets and large binary objects. The relational schema also allows one to define relationships between columns of different tables in the form of foreign key constraints. Even though the relational schema is a powerful description of the data, it has certain limitations in practice. In particular, it cannot accurately describe relationships between columns in the form of *attributes*, i.e., *strongly connected sets of values that appear to have the same or similar meaning within the context of a particular database instance.*

For example, consider a database instance that contains columns about telephone numbers and social security numbers. All such columns can be declared using the same primitive data type (e.g., decimal), but in reality there is never a case where these two types of columns need to be joined with each other: semantically, there is no reason why these columns should belong to the same type. Even though this fact might be known to users (or easy to deduce), it is nowhere explicitly specified within the schema. As another example, consider a database instance that contains a table of customer names and defines two views, one with European and one with Asian customers. Ostensibly, the customer name columns in the European and Asian views will not have any (or very few) values in common. Nevertheless, all three customer name columns belong to the same attribute. Moreover, suppose that there exists a fourth column that contains nation names. Clearly, nation names should not be classified in the same attribute as customer names even though these columns contain the same types of values (i.e., strings). Differentiating between these fundamentally different attributes can be an invaluable tool for data integration and schema matching applications, and, generally speaking, for better understanding and working with the data.

Existing schema matching techniques for identifying relationships between columns use simple statistics and string-based comparisons, e.g., prefix/suffix tests, edit distance, value ranges, min/max similarity, and mutual information based on q-gram distributions [35, 26, 27, 40, 43]. Other approaches use external information like thesauri, standard schemas, and past mappings. Our work on discovering attributes can be used as a valuable addition to all of the above, for designing automated schema mapping tools.

It is important to note here that attribute relationships are not always known in advance to database designers, so it is not always possible to encode them a priori (for example, by using constraints or object oriented schema languages). Certain relationships might hold solely for a particular database instance, others develop over time as the structure of the database evolves, yet others are obvious in hindsight only. Furthermore, there exists a large number of legacy databases (sometimes with sizes in the order of hundreds of tables and thousands of columns) for which schema definitions or folklore knowledge of column meanings might have been lost. To make matters worse, in many practical situations users have access only to a keyhole view of the database (due to access privileges). In such cases users access the data through materialized views, without any information about the underlying schema, or even about the view definitions. In other words, as far as the user is concerned, all schema information has been lost.

Our approach for discovering attributes is purely data oriented. We do not examine solutions that depend on external knowledge about the data. We compute various statistical measures between all pairs of columns within the database, and derive positive and negative relationships between certain pairs of columns. Viewing the database instance as a graph where every column is a node and every positive/negative relationship is an edge, we decompose the graph into connected components. Then, we further decompose each component into a set of attributes.

In particular, in order to discern the type of relationship between any pair of columns, we use Earth Mover's Distance to find the similarity between the distributions of the values contained in the columns. We introduce two types of connections, one based on the *overall distribution* of values and one based on the *intersection distribution* (distribution with respect to the common values only). Low distribution similarity strongly suggests no attribute ties. High intersection distribution similarity suggests very strong attribute ties. We also propose the notion of a *witness column* for introducing relationships by indirect association (i.e., for columns that have no values in common directly, but share

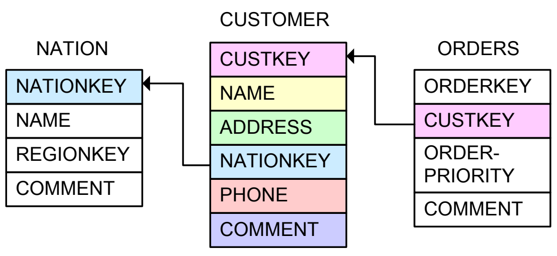


Figure 4.1: Excerpt of the TPC-H schema.

a lot of values with the witness column).

To summarize, our main contributions in this chapter are as follows:

- We provide a robust, unsupervised solution that reports a clustering of columns into attributes.
- We perform a comprehensive empirical study using real and synthetic datasets to validate our solution, and show that it has very high precision in practice.

4.2 Preliminaries

Conventionally, in relational database terminology the term attribute is a synonym for a column. In this work, we use the term attribute to refer to a much stronger notion, based on the *actual meaning* of the values contained in a column. Formally:

Definition 4.1 (Attribute)**.** *An* attribute *is a set of relational columns, such that columns in the same attribute are semantically equivalent to each other.*

In other words, an attribute is a logical notion based on common properties and characteristics of the values contained in the columns comprising that attribute.

For example, Figure 4.1 shows an excerpt of the schema of the TPC-H benchmark [55], which models a business environment and contains information about products, suppliers, customers, orders, etc. The figure shows three tables, CUSTOMER, NATION and ORDERS, and foreign-primary key relationships between some columns of these tables. A customer is associated with six columns in this example: CUSTKEY, NAME, ADDRESS, NATIONKEY,

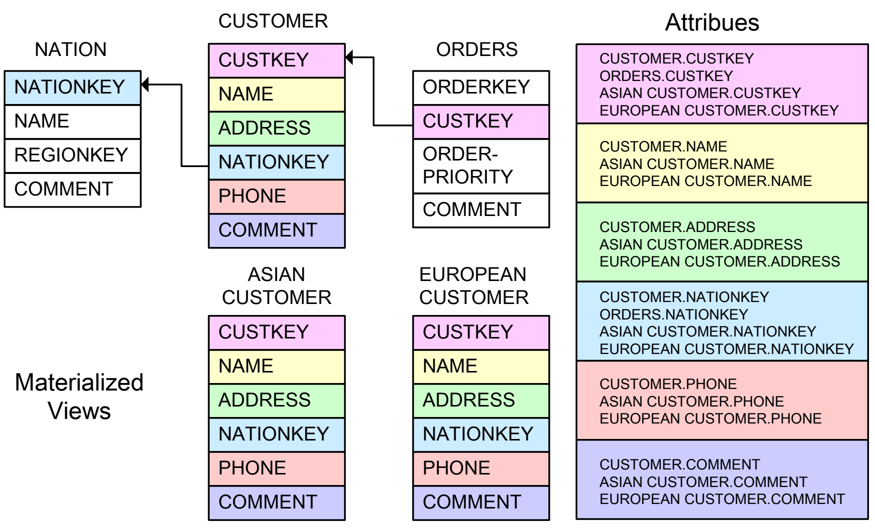


Figure 4.2: Attributes in TPC-H example, which contains three base tables and two materialized views of CUSTOMER table.

PHONE and COMMENT. Since CUSTOMER.NATIONKEY is a foreign key of NATION.NATIONKEY, the two NATIONKEY columns are by definition semantically equal and hence they belong to the same attribute. The same is true for ORDERS.CUSTKEY and CUSTOMER.CUSTKEY. Another example appears in Figure 4.2 which shows a slightly more complex scenario that considers the existence of materialized views, i.e., ASIAN CUSTOMER and EUROPEAN CUSTOMER created from the CUSTOMER table based on NATIONKEY. The ideal clustering of the six columns contained in the CUSTOMER table is shown on the right side of the figure. Clearly, all columns from the three related tables belong to the same attribute, even if there is no direct association specified in the schema (e.g., in the form of primary/foreign keys) and despite the fact that, probably, EUROPEAN CUSTOMER and ASIAN CUSTOMER have no values in common.

We now formalize the problem of attribute discovery as follows:

Definition 4.2 (Attribute Discovery). *Given a collection of relational tables, denoted* $\mathbf{T}$*, let* $\mathbf{C}$ *be the set of all columns in* $\mathbf{T}$*.* Attribute Discovery *is the process of partitioning* $\mathbf{C}$ *into* m *clusters* $\mathbf{A} = \{A_1, A_2, \ldots, A_m\}$ *such that each* $A_k = \{C_1^k, C_2^k, \ldots, C_{n_k}^k\}$ *is an attribute with respect to the set of tables* $\mathbf{T}$*.*

Table 4.1 summarizes the notations used in this chapter. According to Definition 4.1, two columns C and C' are part of the same attribute if and

Symbol	Description
$\mathbf{T}$	Set of tables
$\mathbf{C}$	Set of columns
$\mathbf{A}$	Set of attribute clusters
$\mathbf{N}_C$	Neighborhood of C
$\mathbf{DC}$	Distribution cluster
G_D	Distribution graph
G_A	Attribute graph
ϕ_C	Cutoff value for C
θ	Global threshold for computing ϕ_C

Table 4.1: Notation used throughout Chapter 4.

only if semantically they behave the same. The semantics of two columns can be inferred by the type of relationship these columns have within a database instance. We define the following relationship types:

1. A primary/foreign key;

2. Two foreign keys referring to the same primary key;

3. A column in a view and the corresponding column in the base table;

4. Two columns in two views but from the same column in the base table;

5. No explicit relationship but semantically equivalent (e.g., non-key, customer name columns from different tables).

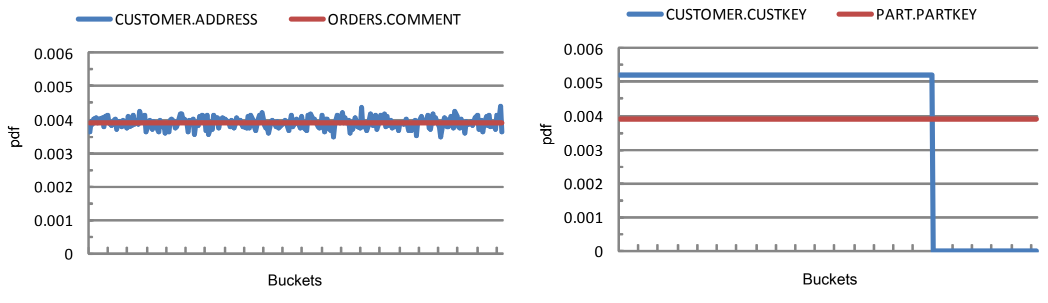
The first four relationship types are, by definition, indicators of strong attribute ties. The fifth relationship type encompasses all columns that are semantically equivalent where this information cannot be inferred from the database schema, but only from the actual values contained in the columns. Only relationship type 1 has been studied in the past. To the best of our knowledge no previous work has studied relationship types 2-5. Nevertheless, existing work can easily be adapted to identify types 2-4. In what follows, we list a set of existing techniques that can be used to identify pairs of columns belonging to these relationship types. In each case, we point out why they are insufficient for identifying *all* relationship types - particularly type 5.

4.2.1 Name Similarity

It is natural to consider using the similarity of column names to infer column semantics, since, to a certain extent, names reflect the meaning of the values within a column. Indeed, previous work, especially in the area of schema matching, has applied this technique to identify associations between columns [44]. However, this is not always a robust solution for three reasons. First, a given database might not use strict naming conventions. Second, columns with no meaningful associations oftentimes have similar or even identical names. For instance, a column called NAME appears in both the NATION and CUSTOMER tables of TPC-H, even though the two columns refer to two semantically unrelated concepts. Third, two semantically related columns may happen to have very different names. For example, the columns in a view might have completely different names from the source columns in the base table. This happens when views are generated automatically, or when long, representative column names have to be abbreviated due to length constraints (e.g., the 30 characters limit in Oracle). Hence, simply relying on column name similarity can lead to both false positives and false negatives, for the purpose of discovering attributes.

4.2.2 Value Similarity

Another straightforward technique is to consider the similarity of the data values contained in a set of columns. The Jaccard coefficient $J(C_1, C_2) = \frac{|C_1 \cap C_2|}{|C_1 \cup C_2|}$ (or any other set similarity measure) can be used for this purpose, which can be efficiently estimated in practice [18]. However, this idea has its own drawbacks. For example, in our TPC-H database instance, column CUSTOMER.CUSTKEY contains values from 1 to 150,000, while column PART.PARTKEY contains all integers from 1 to 200,000. The overlap of the values in these two columns is very high: their Jaccard coefficient is 0.75. Nevertheless, the columns are not semantically related. Conversely, two columns can have a strong semantic relationship but no common values at all, e.g., EUROPEAN CUSTOMER and ASIAN CUSTOMER. Of course, one could argue that in this case the two columns belong to two different attributes. However, in our solution we would like to cluster these columns primarily as customer names (as opposed to, for example, nation names) and, optionally, also partition them into sub-attributes. In this case, using data value similarity alone would lead to a false dismissal.



(a) (b)

Figure 4.3: Data distribution histograms of two examples from TPC-H.

4.2.3 Distribution Similarity

Data distribution similarity has been used in Chapter 3 to discover meaningful primary/foreign key constraints using Earth Mover's Distance (EMD). Computing distribution similarity is a necessary step in our setting for attribute discovery, since we can use it to discover primary/foreign key constraints (one of the core relationship types we are interested in). However, it is not sufficient for the purpose of discovering attributes. Consider for example the values in CUSTOMER.ADDRESS and ORDERS.COMMENT. If we sort them in lexicographic order for the purpose of computing EMD, they follow very similar distributions. The proportion of strings from one column that fall within a given range of strings from the other column in lexicographic order is very similar. To illustrate this point, we plot the two distributions in Figure 4.3(a). The buckets are constructed using an equi-depth histogram based on the quantiles of column ORDERS.COMMENT. Then, we simply tally the number of strings from column CUSTOMER.ADDRESS that fall within each bucket. The plot clearly shows that the values in CUSTOMER.ADDRESS also follow a nearly uniform distribution across buckets. Indeed, the EMD between the two columns is only 0.0004. Still, computing distribution similarity does eliminate a large number of other column pairs: for example, the histograms for columns CUSTOMER.CUSTKEY and PART.PARTKEY, whose EMD is 0.125, are shown in Figure 4.3(b). We conclude that EMD values are a useful but insufficient filter. The main reason why EMD works for discovering primary/foreign key relationships, but not for discovering attributes, is the fact that for primary/foreign keys a containment relationship needs to hold: By definition, most, if not all, values from the foreign key must belong to the primary key. Thus, we would never consider ORDERS.COMMENT and CUSTOMER.ADDRESS as a valid primary/foreign key candidate in the first place, since they are not expected to have many strings in common. By contrast, for columns belonging to the same attribute no containment relationship needs to hold (e.g., EUROPEAN CUSTOMER and ASIAN CUSTOMER).

4.3 Attribute Discovery

It is clear that simply applying the aforementioned methods for discovering attributes will not yield accurate results. In this section we present a novel two-step approach. Intuitively, most of the time columns that belong to the same attribute tend to contain values that are drawn from the same underlying distribution. Conversely, if the values of two columns have different distributions, they more likely belong to different attributes. Therefore, here we advocate an algorithm that uses data distribution similarity, based on EMD, for partitioning columns into *distribution clusters*. This first step is used to separate columns into major categories, for example clusters that contain strings and clusters that contain only numerical columns. Furthermore, this step will also separate numerical columns with widely differing distributions, based for example on the range of values within the columns.

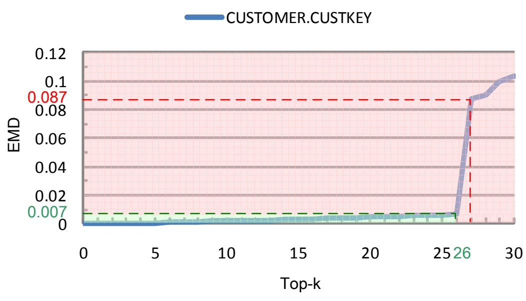
The use of EMD to create distribution clusters has some limitations. First and foremost, as can be seen from the example in Figure 4.3(a), not all the columns in one distribution cluster belong to the same attribute, especially when it comes to columns containing strings. String columns tend to have very similar distributions irrespective of the attribute these strings are derived from (e.g., addresses and comments). Second, using EMD might place some columns that belong to the same attribute into different clusters. This will happen for example if several views are defined on a table containing regions and telephone numbers, and the views select telephone numbers only from particular regions. Clearly, the telephone numbers in each view will start with a particular prefix. These correlations result in significantly different distribution of values between columns of the same attribute, rendering distribution based measures ineffective.

To solve the first problem, we use a subsequent, refinement phase that relies on computing the similarity of the distribution based on the intersection of two columns. We also use indirect associations through witness columns for cases where two columns have no values in common. We leave the solution of the second problem as future work, and identify this scenario as a limitation of our algorithm. From our experience, columns that have been generated based on indirect correlations do exist, but are rare in practice.

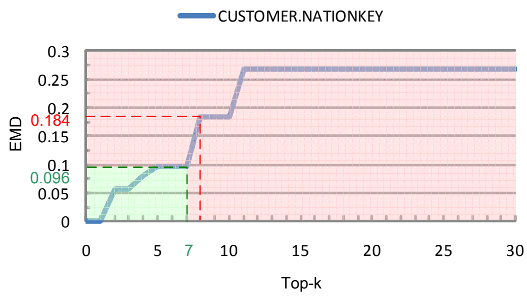
4.3.1 Phase One: Computing Distribution Clusters

Given a set of columns $\mathbf{C}$, we form distribution clusters by computing the EMD between all pairs of columns in $\mathbf{C}$. Since EMD is a symmetric measure, this step requires $\frac{|\mathbf{C}|(|\mathbf{C}|-1)}{2}$ EMD computations. Then, given all pairwise EMDs we need to decide how to partition columns into clusters. The main idea here is to derive for every column a set of neighboring columns with small EMD, such that no two columns that intuitively belong to the same attribute are ultimately split into separate clusters. For every column C, we sort its EMD values to every other column in increasing order. Now we have to choose a cutoff EMD threshold that will determine the neighborhood $\mathbf{N}_C$ of C. After all neighborhoods have been determined, we will form a graph G_D, where nodes correspond to columns and undirected edges are created between a column C and all other columns in $\mathbf{N}_C$. Finally, the distribution clusters are formed by computing all connected components in G_D.

In practice, a small EMD value is not only subjective, but also attribute dependent. From our experience, different types of attributes exhibit significantly different behavior in terms of the similarity of the distribution of values across columns of that attribute, even when they are of the same primitive data type. We illustrate this with an example from TPC-H in Figure 4.4. For columns CUSTOMER.CUSTKEY and NATION.NATIONKEY, we sort their EMD values to all other columns in the database in increasing order of EMD, and plot the top-30 results. From careful, manual analysis of the data, we have determined that the columns belonging to the same attribute as CUSTKEY and NATIONKEY fall in the green region of the plots. We can clearly see that both the cutoff EMD value and the value of k that bound the green regions in the plots, differ significantly for these two attributes. For our purposes we would like to be able to identify the EMD cutoff threshold for each column automatically (and not by choosing a global EMD threshold or value k), and clearly this means that



(a)



(b)

Figure 4.4: EMD plot of two examples in TPC-H.

we have to resort to heuristics.

We observe an interesting property that holds for most columns, in all test cases that we have examined. Given a column C and all pairwise EMD values, it appears that there usually exists a big gap in the magnitude of EMD values, in the sorted order. For example, in both Figures 4.4(a) and 4.4(b) we observe a big gap in the sorted EMD order, after the cutoff region (the dotted red lines in the plots). Intuitively, the gap signifies a cutoff point below which other columns seem to have similar distributions with C, and above which columns seem to be completely unrelated to C (recall that small EMD means *similar* and large EMD means *different*). This is expected for most column types in realistic scenarios. For example, numerical columns are more similar to each other than string columns (hence, a big gap exists in the EMD values where numerical columns end and string columns begin); even among numerical columns, smaller gaps occur because of the different ranges of values in the columns (e.g., salaries vs. zip codes). Conservatively choosing this gap to be the cutoff EMD threshold that defines the neighborhood $\mathbf{N}_C$ guarantees that most false cluster splits are avoided in practice (in other words we do not end up with too many clusters). In addition, we can also use a conservative global EMD cutoff threshold θ (a value large enough to signal that two distributions are significantly different) to make sure that the opposite problem does not occur either, i.e., we do not end up with too few distribution clusters.

Algorithmically identifying the cutoff EMD threshold for a column C is straightforward. Let θ be the global threshold, and $L(C)$ be the sorted list of EMD values for C, i.e., $L(C)$ contains all values $e = EMD(C, C')$, $\forall C' \in \mathbf{C}$, in increasing order. We truncate $L(C)$ to values smaller than θ and identify the largest difference between two consecutive EMD values in the truncated list. The pseudo code appears in Algorithm 4.1. In the algorithm we also add θ in the list of EMD values to capture the special case were the largest gap between two values happens to involve θ.

Algorithm 4.1: ComputeCutoffThreshold ($L(C)$, θ)

```
1 $L = L \cup (\theta, \emptyset)$
2 Sort $L$ in increasing order of EMD values
3 $\phi_C = 0$, $i = 0$, $gap = 0$
4 while $L[i+1] \leq \theta$ do
5     if $gap < L[i+1].e - L[i].e$ then
6         $gap = L[i+1].e - L[i].e$
7         $\phi_C = L[i].e$
8     $i = i + 1$
9 Return $\phi_C$
```

Algorithm 4.2: ComputeDistributionClusters ($\mathbf{C}$, θ)

```
1  $G_D = \emptyset$
2  for $i \leftarrow 1$ to $|\mathbf{C}|$ do
3      for $j \leftarrow i+1$ to $|\mathbf{C}|$ do
4          $e = EMD(C_i, C_j)$
5          $A[C_i].insert(e, C_j)$      /* $A$ is a hash table of $(e, C)$ */
           $A[C_j].insert(e, C_i)$
6      $G_D.AddNode(C_i)$
7  for $i \leftarrow 1$ to $|\mathbf{C}|$ do
8      $\phi_{C_i} = ComputeCutoffThreshold(A[C_i], \theta)$
9      foreach $C_j \in \mathbf{N}_{C_i}$ do
10         $G_D.AddEdge(C_i, C_j)$
11 Return connected components of $G_D$
```

Once the cutoff value ϕ_C for each column C has been computed we can define the neighborhood of a column as follows:

Definition 4.3 (Neighborhood)**.** *The neighborhood* $\mathbf{N}_C$ *of column* C *consists of all columns* C' *with* $EMD(C, C') \leq \phi_C$*.*

Then, we build the *distribution graph*, which is defined as follows:

Definition 4.4 (Distribution Graph)**.** *A Distribution Graph* $G_D = (V_D, E_D)$ *is an undirected graph where each column* $C \in \mathbf{C}$ *corresponds to a node* $C \in V_D$*, and an edge between nodes* C *and* C' *exists iff* $C \in \mathbf{N}_{C'} \vee C' \in \mathbf{N}_C$*.*

Alternatively, we can define the edges as $C \in \mathbf{N}_{C'} \wedge C' \in \mathbf{N}_C$, but our experimental evaluation shows that in practice this does not affect precision.

The distribution clusters are obtained by computing the connected components in the resulting distribution graph:

Definition 4.5 (Distribution Cluster)**.** *Let* $G_i = (V_D^i, E_D^i), V_D^i \subset V_D, E_D^i \subset E_D, 1 \leq i \leq n$ *be the set of* n *connected components in distribution graph* G_D*. The set of columns corresponding to the nodes in* V_D^i *determines distribution cluster* $\mathbf{DC}_i$*.*

The pseudo code for computing distribution clusters is shown in Algorithm 4.2. We can compute the connected components of graph G_D (line 13 of Algorithm 4.2) using either depth-first or breadth-first search.

Figure 4.5 shows the distribution clusters of the TPC-H example in Figure 4.2, which contains three base tables and two materialized views (table names EUROPEAN CUSTOMER and ASIAN CUSTOMER are shortened to EC and AC). Using the distribution graph, twenty six columns are partitioned into eight clusters. Columns from distinctly different domains are immediately separated (e.g., numeric values and strings). The numeric columns with different ranges of values are also correctly clustered (e.g., key columns like CUSTKEY, NATIONKEY, ORDERKEY and REGIONKEY), as well as columns that contain specially formatted values (e.g., PHONE which contains numerals and dashes). However, distribution clusters cannot always differentiate between different string columns (e.g., ADDRESS and COMMENT).

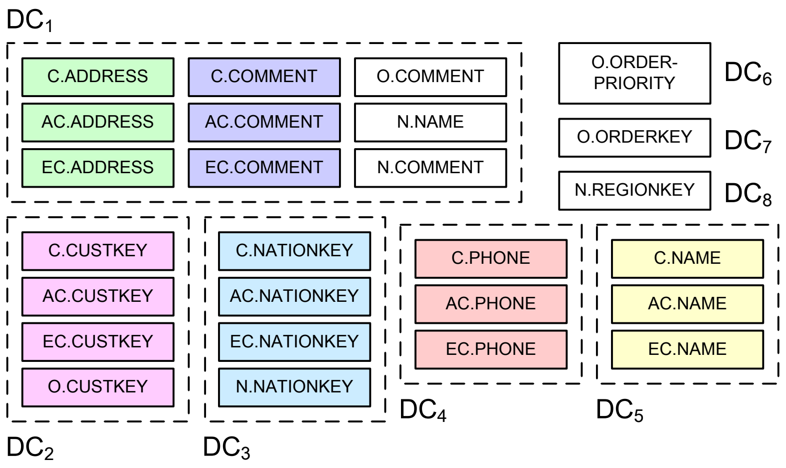


Figure 4.5: Distribution clusters of TPC-H example.

4.3.2 Phase two: Computing Attributes

We now describe in detail how to further decompose distribution clusters into attributes, specifically for identifying columns that have very similar distributions overall but do not belong to the same attributes, as is the case for many string columns. We use an intersection distribution metric and witness columns, to construct one attribute graph per distribution cluster and then correlation clustering to decompose the cluster into attributes.

The Attribute Graph

In order to decompose clusters into attributes we create one *attribute graph* G_A per cluster. Given that all columns within the same distribution cluster have similar distributions of values, we need to differentiate between attributes by also taking into account the values these columns have in common. Clearly, columns with large intersection of values are highly related and should belong to the same attribute (this is similar to automatically identifying whether two columns have a primary/foreign key relationship, as in Chapter 3). On the other hand, columns that have very few or no values in common, might or might not belong to the same attribute (e.g., as is the case of EUROPEAN CUSTOMER and ASIAN CUSTOMER, and conversely, ADDRESS and COMMENT).

We make here the following key observation. We can determine whether two columns with empty intersection come from the same attribute by using a *witness* column, i.e., a third column that is highly related to both columns. In other words, we introduce *relationships by indirect association*. For example, we know for a fact that both EUROPEAN CUSTOMER and ASIAN CUSTOMER have a large number of values in common with CUSTOMER, but not with each other. After identifying that CUSTOMER is related with EUROPEAN CUSTOMER and ASIAN CUSTOMER, we can deduce with high confidence that EUROPEAN CUSTOMER and ASIAN CUSTOMER are also related. Formally:

Definition 4.6 (Witness Column). *Consider three distinct columns C, C', C''. Column C'' is a witness for C and C' if and only if both conditions hold:*

1. *C'' and C are in the same attribute.*
2. *C'' and C' are in the same attribute.*

Clearly, if two columns belong to the same attribute, have no values in common, and no witness column, then we will not be able to identify these columns. This is one more limitation of our approach, but in practice, such cases might either be identifiable using orthogonal techniques (e.g., column name similarity), or in other cases might be hard to identify using any unsupervised solution.

Based on these observations, we create the *attribute graph* of each cluster **DC**, similarly to the distribution graph of **C**. Here, a node corresponds to a column of **DC** and an edge corresponds to an intersection relationship between two columns.

We also have to define a measure over these edges, which we call Intersection EMD. Intersection EMD measures the likelihood that a pair of columns are part of the same attribute, taking into account the distribution of the values within each column with respect to the *common* values. In general, for an edge (C, C'), $EMD(C, C \cap C') \neq EMD(C', C \cap C')$. Since the edge is undirected, we define its weight as the arithmetic mean of these two values. Formally:

Definition 4.7 (Intersection EMD). *Given columns C, C', the Intersection EMD is defined as:*

$$EMD_{\cap}(C, C') = \frac{1}{2}(EMD(C, C \cap C') + EMD(C', C \cap C')).$$

Let $EMD_{\cap}(C, C') = \infty$, if $C \cap C' = \emptyset$.

Clearly, Intersection EMD can differentiate between columns like ADDRESS and COMMENT, since their intersection is empty. Even if they did have a small number of values in common, their Intersection EMD would be very large.

Similar to the case of computing distribution clusters, we need to decide whether the Intersection EMD between two clusters is small enough to place the columns into the same attribute. We are trying to balance the number of attributes to create (small thresholds will result in too many attributes and large thresholds in too few). For each individual column, we compute a cutoff threshold as before (see Algorithm 4.1), but instead of using EMD we use Intersection EMD. Similarly, we define the neighborhood $\mathbf{N}_C$ of a column C, this time with respect to Intersection EMD.

We now give the formal definition of the attribute graph corresponding to a distribution cluster:

Definition 4.8 (Attribute Graph)**.** *The attribute graph $G_A = (V_A, E_A)$ of a distribution cluster* **DC** *is a* complete graph *over the set of vertices of* **DC***, such that the weights of edges in E_A are either 1 (positive edges) or -1 (negative edges). Let E_A^+, E_A^- denote the set of positive, resp. negative, edges in G_A. To define them, consider an arbitrary pair of vertices $C, C' \in V_A$.*

1. *Neighborhood: If $C \in \mathbf{N}_{C'} \vee C' \in \mathbf{N}_C$, then $e_{CC'} \in E_{A1}^+$.*
2. *Witness: If $\exists C'' \in V_A$ s.t. $e_{CC''} \in E_{A1}^+ \wedge e_{C'C''} \in E_{A1}^+$, then $e_{CC'} \in E_{A2}^+$.*

We define $E_A^+ = E_{A1}^+ \cup E_{A2}^+$, and $E_A^- = E_A \setminus E_A^+$.

Figure 4.6 shows the attribute graph of distribution cluster $\mathbf{DC}_1$ from Figure 4.5. The green lines in the figure denote positive edges while the red lines are negative edges. The edges between the three nodes outside the dashed box to all other nodes are negative. Using Intersection EMD we are able to separate columns like ADDRESS and COMMENT, while AC.ADDRESS and EC.ADDRESS are connected through the witness column C.ADDRESS. The same holds for AC.COMMENT and EC.COMMENT.

The next step is to decompose the graph into attributes. Clearly, we could decompose the graph into connected components (by simply ignoring negative edges) similarly to phase one. Nevertheless, due to the nature of Intersection EMD and the fact that after phase one, attribute graphs consist of a small

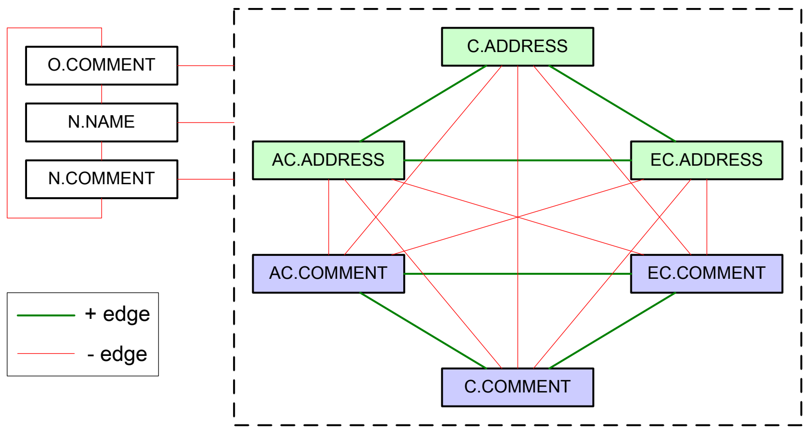


Figure 4.6: A possible attribute graph of distribution cluster $\mathbf{DC}_1$.

number of nodes, in practice attribute graphs tend to comprise of a single (or very few) connected components. A better approach here is to use the negative weights productively to find an optimal clustering of nodes into attributes that minimizes the number of conflicting nodes that end up into the same cluster and the number of related nodes that end up in different clusters. As it turns out, this is exactly the goal of *correlation clustering*.

Correlation Clustering

Let $G = (V, E)$ be an undirected graph with edge weights 1 or -1. Let E^+ be the set of positive edges, and E^- be the set of negative edges; $E = E^+ \cup E^-$. Intuitively, edge $e_{uv} \in E^+$ if u and v are similar; and $e_{uv} \in E^-$ if u and v are dissimilar. The correlation clustering problem [9] on G is defined as follows:

Definition 4.9 (Correlation Clustering)**.** *Compute disjoint clusters covering V, such that the following cost function is minimized:*

$$\begin{aligned} cost \quad = \quad & |\{e_{uv} \in E^+ \mid Cl(u) \neq Cl(v)\}| + \\ & |\{e_{uv} \in E^- \mid Cl(u) = Cl(v)\}|, \end{aligned}$$

where $Cl(v)$ denotes the cluster node v is assigned to.

This definition minimizes the total number of disagreements, i.e., the number of positive edges whose endpoints are in different clusters, plus the number of negative edges whose endpoints are in the same cluster. Alternatively, one can define the dual problem of maximizing the total agreement. More general versions of the problem exist, e.g., when weights are arbitrary real values. However, this version is sufficient for our purposes.

Correlation clustering can be written as an integer program, as follows. For any pair of vertices u and v, let $X_{uv} = 0$ if $Cl(u) = Cl(v)$, and 1 otherwise. The integer program is

Minimize:

$$\sum_{e_{uv} \in E^+} X_{uv} + \sum_{e_{uv} \in E^-} (1 - X_{uv})$$

s.t.

$$\forall u, v, w : X_{uw} \leq X_{uv} + X_{vw}$$

$$\forall u, v : X_{uv} \in \{0, 1\}$$

The condition $X_{uw} \leq X_{uv} + X_{vw}$ ensures that the following transitivity property is satisfied: if $X_{uv} = 0$ and $X_{vw} = 0$, then $X_{uw} = 0$ (note that this is equivalent to: if $Cl(u) = Cl(v)$ and $Cl(v) = Cl(w)$, then $Cl(u) = Cl(w)$). Therefore X defines an equivalence relationship, and the clusters are its equivalence classes. Correlation clustering is NP-Complete [9]. Nevertheless, the above integer program can be solved exactly by IP solvers (e.g., CPLEX [1]) for sufficiently small graphs. For larger graphs, one can use polynomial time approximation algorithms [8].

Going back to the example of Figure 4.6, correlation clustering on this graph will further decompose nine columns into five attributes, as shown in Figure 4.7. If all the edges in the attribute graph are correctly labeled, such as the simple example in Figure 4.6, then the graph results in perfect clustering, meaning that there are no disagreements. (When this is the case, simply removing all the negative edges and computing the connected components in the remaining graph also returns the correct attributes.) However, if a few edges are assigned conflicting labels, there is no perfect clustering. For example, Figure 4.8 shows a different attribute graph for distribution cluster DC_1, obtained by setting a higher threshold θ in Algorithm 1. The edge between AC.ADDRESS and AC.COMMENT is now labeled positive. In addition, two other positive edges are created, since AC.ADDRESS and AC.COMMENT act as witnesses for C.ADDRESS and C.COMMENT. As it turns out, in this case correlation clustering is still able to separate the columns correctly, by finding

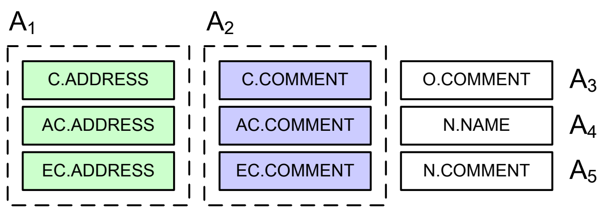


Figure 4.7: Attributes discovered in the attribute graph of distribution cluster $\mathbf{DC}_1$.

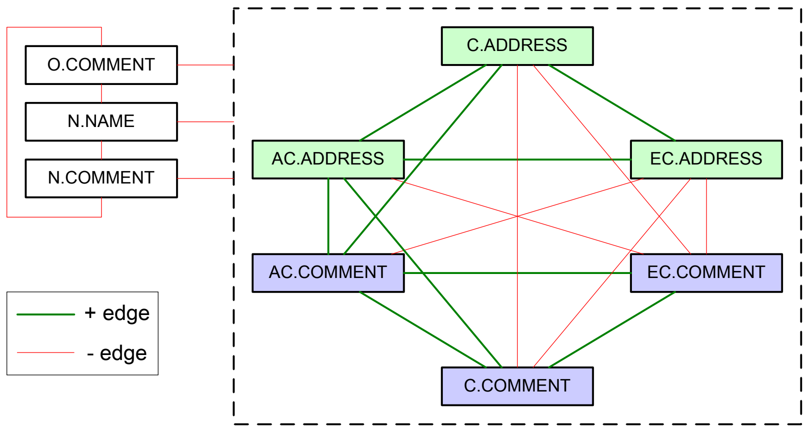


Figure 4.8: Another possible attribute graph of distribution cluster $\mathbf{DC}_1$.

a partition that agrees as much as possible with the edge labels. Of course, in some cases correlation clustering will result in mistakes, but in the end our solution will decompose the graph into attributes that can be manually inspected much more easily than having to look at the complete distribution graph.

We now summarize phase two. The pseudo code is shown in Algorithm 4.3. For each distribution cluster computed in phase one, compute the Intersection EMD between each pair of columns in the cluster and store the resulting values in a hash table I in increasing order of Intersection EMD. Then compute the cutoff threshold for each column. Construct the attribute graph G_A according to Definition 4.8. Creating positive edges based on witness columns is implemented by creating positive edges between nodes with path of length two. This is accomplished by first computing the adjacency matrix E of graph $G_{A1} = (V_A, E^+_{A1})$, where $E[C_i][C_j] = 1$ means the edge between node C_i and C_j is positive. The adjacency matrix of graph $G_A = (V_A, E^+_{A2})$ can be computed by multiplying E by itself. The sum of the two matrices is the final adjacency matrix M of attribute graph $G_A = (V_A, E_A)$. Finally, we compute attributes using correlation clustering on graph G_A.

Algorithm 4.3: ComputeAttributes ($\mathbf{DC}$, θ)

```
1  $G_A = \emptyset$, $E[][] = 0$, $M[][] = 0$
2  for $i \leftarrow 1$ to $|\mathbf{DC}|$ do
3      for $j \leftarrow i+1$ to $|\mathbf{DC}|$ do
4          $e = EMD_{\cap}(C_i, C_j)$
5          $I[C_i].insert(e, C_j)$      /* $I$ is a hash table of $(e, C)$ */
           $I[C_j].insert(e, C_i)$
6      $\phi_{C_i} = ComputeCutoffThreshold(I[C_i], \theta)$
7      foreach $C_j \in \mathbf{N}_{C_i}$ do
8          $E[C_i][C_j] = 1$
9      $G_A.AddNode(C_i)$
10 $M = E + E \times E$
11 for $i \leftarrow 1$ to $|\mathbf{DC}|$ do
12     for $j \leftarrow 1$ to $|\mathbf{DC}|$ do
13         if $M[i][j] = 0$ then
14             $G_A.AddNegativeEdge(C_i, C_j)$
15         else
16             $G_A.AddPositiveEdge(C_i, C_j)$
17 Return correlation clustering of $G_A$
```

4.4 Performance Considerations

Clearly, due to the difficult and subjective nature of this problem, no unsupervised solution will lead to 100% precision 100% of the time. The solution provided here can be used in conjunction with other techniques for improving quality. Notice that the two phases of our algorithm are quite similar. We create a graph based on some similarity measure and decompose the graph based on connected components or correlation clustering. The heuristic nature of the algorithm raises a number of questions about possible alternative strategies. For example, we could reverse the two steps, or use Intersection EMD instead of EMD first. We can also use correlation clustering in the first phase of the algorithm.

We use EMD first simply because it is a weaker notion of similarity than Intersection EMD. EMD acts upon all values of two columns, while Intersection

EMD acts upon the common values. EMD is used to roughly decompose the instance graph into smaller problems, by separating columns that clearly belong to different attributes; strings from numerical columns and columns with significantly different ranges and distributions of values. The graph based on EMD edges alone (without any Intersection EMD edges) is sparse and easy to partition into smaller instances. Of course, we could combine both phases into one by creating a graph with EMD, Intersection EMD and witness edges, but we use a two phase approach here for efficiency. For the same reason we do not use correlation clustering in the first phase. Running correlation clustering on the distribution graph G_D can be very expensive due to the large number of nodes. On the other hand, running correlation clustering independently on the much smaller connected components is manageable.

Notice that the cost of computing EMD and Intersection EMD depends on the size of the columns involved. Clearly, for columns containing a very large number of distinct values the cost of computing all pairwise EMD and Intersection EMD values can be prohibitive. For that reason we can approximate both measures by using the technique proposed in Chapter 3, which is based on quantiles. We shall briefly describe the technique here for explanation purpose. Essentially the technique computes a fixed number of quantiles from all columns (e.g., 256 quantiles) and then computes EMD between two columns by using the quantile histograms. In particular, given two columns C and C', $EMD(C, C')$ is computed by taking the quantile histogram of C and performing a linear scan of C' to find the number of values in C' falling into each bucket of the histogram of C (notice that we cannot compute EMD between two quantile histograms directly, since the bucket boundaries might not coincide, in which case EMD is undefined). Then $EMD(C, C')$ is approximated as the EMD between the two resulting histograms. The intuition here is that quantile summaries are a good approximation of the distribution of values in the first place, hence the EMD between two quantile summaries is a good approximation of the EMD between the actual columns. Moreover, this approach has a proven bounded error of approximation, as shown in Chapter 3.

Computing Intersection EMD entails computing the intersection between all pairs of columns, which of course is a very expensive operation, especially if no index exists on either column. In order to improve performance we build Bloom filters [13] and use the Bloom filters to compute an approximate intersection between sets. Given two columns C and C', we first perform a linear scan of column C, probe the Bloom filter of C', and if the answer is positive, use the existing quantile histograms of C and C' to (approximately) compute both $EMD(C, C \cap C')$ and $EMD(C', C \cap C')$. Optionally, we can improve the approximation of the intersection by also scanning column C' and probing the Bloom filter of C. Given that Bloom filters introduce false positives, this approach can result in cases where two columns have an empty intersection, but their approximate Intersection EMD is finite. Nevertheless, for columns of very large cardinality (especially in the absence of indexes), using Bloom filters results in significant performance improvement. One can further balance accuracy and performance, by using exact intersection computations for small columns, and Bloom filters for large columns.

As discussed above, correlation clustering is NP-Complete. Nevertheless, we solve it exactly, by running CPLEX [1] on its corresponding integer program. In our experiments, CPLEX was able to find solutions for large graph instances very fast. The largest graph instance we tried contained 170 nodes, 14365 variables, and 2.5 million constraints and took 62 seconds to complete using four Intel IA-64 1.5GHz CPUs and four threads. Alternatively, one can use polynomial time approximation algorithms [8] if faster solutions are required for larger graphs.

4.5 Experimental Evaluation

We conducted extensive experiments to evaluate our approach on three datasets based on the TPC-H [5] benchmark (with scale factor 1), and the IMDB [3] and DBLP [2] databases. For each dataset, we created a large set of materialized views to simulate a more complex scenario. The detailed statistics of all datasets are given in Table 4.2. The views are created from a selection of rows based on the values of a specific column (mostly columns that contain categorical data) and each of the views represents a semantically meaningful subset of the data in the base table. Table 4.3 summarizes the views generated in each dataset. The experiments were run on an Intel Core2 Duo 2.33 GHz CPU Windows XP box and CPLEX was run on a four Intel IA-64 1.5GHz CPU Linux box.

We use two standard metrics, *precision* and *recall*, to measure the quality of discovered attributes. The gold standard was manually identified from a

	Base tables	Views	Columns	Rows
TPC-H	8	110	785	12,680,058
IMDb	9	118	254	12,048,155
DBLP	6	66	285	8,647,713

Table 4.2: Datasets statistics.

Dataset	Base Table	View No.	Selection	Description
TPC-H	CUSTOMER	1-2	ACCTBAL	Customers with positive/negative account balance.
		3-7	MKTSEGMENT	Customers in each market segment.
		8-37	NATIONKEY	Customers from each nation/region.
	NATION	38-42	REGIONKEY	Nations in each region.
	PART	43-67	BRAND	Parts of each brand.
		68-72	MFGR	Parts by each manufacture.
	SUPPLIER	73-102	NATIONKEY	Suppliers from each nation/region.
	ORDERS	103-105	ORDERSTATUS	Orders in each status.
		106-110	ORDERPRIORITY	Orders with each priority.
IMDb	MOVIE	1-28	COUNTRY	Movies released in each country.
		29-118	YEAR	Movies released in each year.
DBLP	ARTICLES	1-20	YEAR	Journal papers published in each year.
	INPROCEEDINGS	21-38	YEAR	Conference papers published in each year.
	BOOKS	39-66	YEAR	Books published in each year.

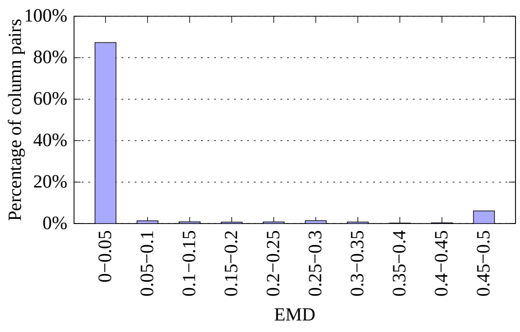
Table 4.3: Description of materialized views.

careful analysis of each dataset. Given the nature of the problem, we define precision as a measure of purity of a discovered attribute (how similar is the set of columns contained in the attribute with respect to the gold standard), and recall as a measure of completeness. Let the set of discovered attributes be $\mathbf{A} = \{A_1, A_2, \ldots, A_m\}$ and the gold standard be $\mathbf{T} = \{T_1, T_2, \ldots, T_{m'}\}$. We first define the precision and recall of a discovered attribute A_i. Each column in A_i belongs to an attribute in $\mathbf{T}$. Let A_i correspond to T_j if and only if the majority of columns in A_i also belong in T_j. Then, the precision and recall of A_i are defined as:

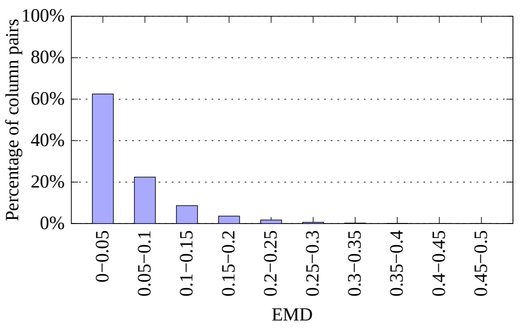
$$Precision(A_i) = \frac{|A_i \cap T_j|}{|A_i|},\ Recall(A_i) = \frac{|A_i \cap T_j|}{|T_j|}.$$

We then define the precision and recall of the final result $\mathbf{A}$ as the average over all attributes:

$$Precision(\mathbf{A}) = \frac{\sum_{i=1}^{m} Precision(A_i)}{m},\ Recall(\mathbf{A}) = \frac{\sum_{i=1}^{m} Recall(A_i)}{m}.$$



(a) TPC-H



(b) DBLP

Figure 4.9: Distribution histograms of EMD values between all pairs of columns in the same attribute for TPC-H and DBLP.

4.5.1 Distribution Similarity

As already discussed, in most cases columns that belong to the same attribute tend to have similar distributions, and columns that have different distributions more likely belong to different attributes. First, we run experiments to verify this intuition. For each dataset, we examine the EMD values between all pairs of columns in the same attribute, based on the gold standard, and plot the distribution histograms (for efficiency we approximate all EMD computations using quantile histograms). Figure 4.9 shows the results for TPC-H and DBLP. For TPC-H 87.3% EMD values between columns of the same attribute are smaller than 0.05. For DBLP 62.5% are below 0.05 and only 2.8% are above 0.2. This verifies our intuition that EMD is a robust measure for phase one of the algorithm.

Notice that a few pairs of columns in TPC-H have very large EMD. This is caused by the four attributes shown in Table 4.4. View1 and View2 select customers with positive and negative balance (see Table 4.3), which results in a horizontal partition of the base table and very different distributions in each partition. The same happens for attributes phone number and order date. Since View8-37 and View73-102 select customers and suppliers from a particular nation/region, the phone numbers in each view start with the same prefixes. View103-105 are the orders in each particular status and order status is correlated to the date when the order is placed. Distribution similarity fails to discover the associations between columns if such correlations exist, and this is a limitation of our approach. However, we can see here that distribution

Attribute	Columns
Customer account balance	ACCTBAL in CUSTOMER and View1-2
Customer phone number	PHONE in CUSTOMER and View8-37
Supplier phone number	PHONE in SUPPLIER and View73-102
Order date	DATE in ORDERS and View103-105

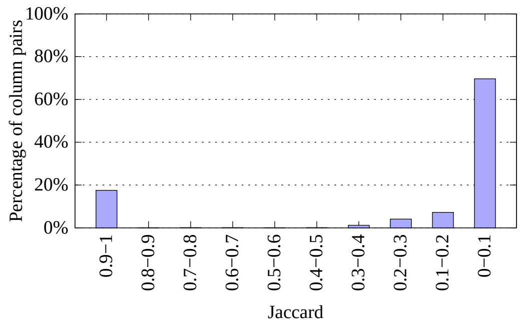
Table 4.4: Attributes that contain horizontally partitioned columns in TPC-H.

similarity between columns of the same attribute holds for the large majority of columns. After removing the horizontal partitions mentioned above from TPC-H (65 columns in total), the EMD values between all pairs of columns within the same attribute are below 0.2 and for up to 99.5% of the pairs, EMD is below 0.05.

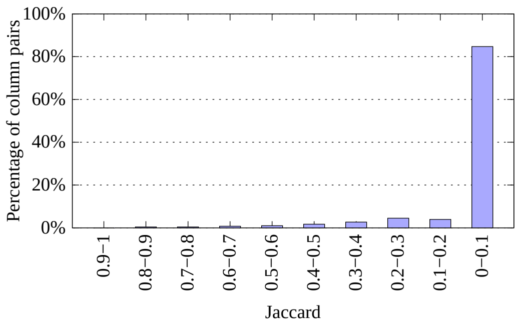
To illustrate the point that columns of the same attribute do not necessarily have too many values in common, in Figure 4.10 we plot a histogram of the pairwise Jaccard similarity of columns within the same attribute, based on the golden standard. Recall that a high Jaccard value implies a large number of common values and vice versa. We observe that for TPC-H 70% of column pairs have Jaccard similarity smaller than 0.1, and only 19% have Jaccard above 0.9. The results for DBLP are even more striking, with more than 80% of column pairs having Jaccard similarity smaller than 0.1. It is thus clear that Jaccard similarity is a poor measure for clustering columns into attributes. In particular, a naive approach for discovering attributes would be to create a column similarity graph with edges weighted according to pairwise Jaccard similarity, then remove edges with Jaccard similarity smaller than some threshold, and finally compute the connected components. Figure 4.10 clearly shows that dropping edges with reasonably small Jaccard similarity would result in a very sparse graph, separating columns into atomic attributes. On the other hand, retaining all edges would tend to separate columns into two attributes, one for numerical attributes and one for strings.

4.5.2 Attribute Discovery

Here we measure the accuracy of our technique for discovering attributes. We use a single global threshold θ for computing the distribution clusters in Phase one as well as building the attribute graph in Phase two. Furthermore, we use Bloom filter for all columns, across the board, to approximate Intersection



(a) TPC-H



(b) DBLP

Figure 4.10: Distribution histograms of Jaccard values between all pairs of columns in the same attribute for TPC-H and DBLP.

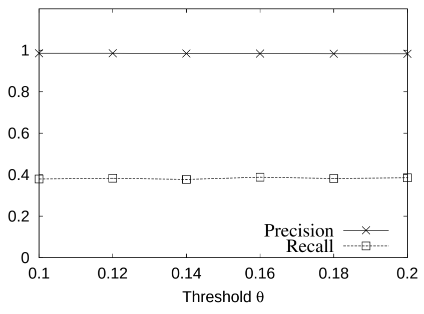
EMD. In the experiments, we vary θ from 0.1 to 0.2. Table 4.5 shows the accuracy results for all datasets. For TPC-H, we report two sets of results. TPCH-1 refers to the results with respect to the original dataset. TPCH-2 refers to the results with respect to a reduced dataset, obtained by removing the horizontally partitioned columns. For readability, we also plot the precision and recall in Figure 4.11. We can see that for large ranges of thresholds θ (0.16-0.2) we achieve high precision and recall for all datasets, which makes our approach easy to use in practice.

For the TPC-H dataset, as already explained, 65 columns (belonging to only 4 attributes out of 45) are from horizontal partitions of the base tables due to indirect correlations between columns. Here, distribution similarity fails to cluster such columns together, more precisely each view becomes its own cluster, resulting in more than 100 attributes overall. However, as shown in TPCH-2, by drilling down we can see that our approach achieves high precision and recall for discovering attributes in the remaining set of columns. It should be noted here that columns that form unit clusters can be singled out and treated separately during post-processing. For future work, we are investigating whether it is possible to identify if unit clusters belonging to horizontally partitioned columns can be concatenated as a subsequent step.

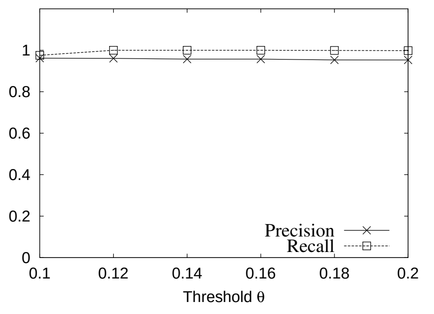
Our approach discovers fewer attributes than the gold standard. Table 4.6 shows the attributes that cause the errors for $\theta = 0.12$. As shown, nine distinct attributes are clustered into four attributes by our algorithm (one attribute per row), and that accounts for the five missing attributes. Here, 9997 out of 10000 values in SUPPLIER.ADDRESS are identical to the values

Threshold θ		0.1	0.12	0.14	0.16	0.18	0.2
TPCH-1	m'	46					
	m	108	107	106	103	102	101
	P	0.986	0.985	0.984	0.984	0.983	0.983
	R	0.379	0.383	0.377	0.388	0.382	0.385
TPCH-2	m'	45					
	m	41	40	39	39	38	38
	P	0.962	0.961	0.957	0.957	0.953	0.953
	R	0.976	1	1	1	0.999	0.998
IMDB	m'	10					
	m	12	11	11	10	10	10
	P	0.958	0.955	0.955	0.95	0.95	0.95
	R	0.75	0.818	0.818	0.9	0.9	0.9
DBLP	m'	14					
	m	19	14	13	12	12	12
	P	0.949	0.93	0.924	0.918	0.918	0.918
	R	0.632	0.857	0.92	0.997	0.997	0.997

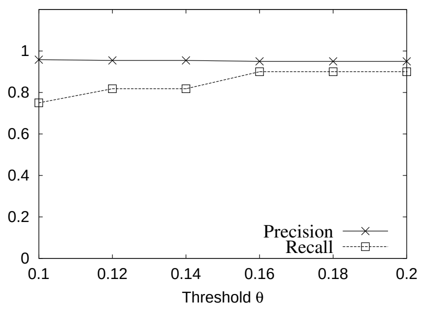
Table 4.5: Accuracy results on TPC-H, IMDB and DBLP for different thresholds θ; m' true number of attributes; m attributes in our solution; P is precision; R is recall.



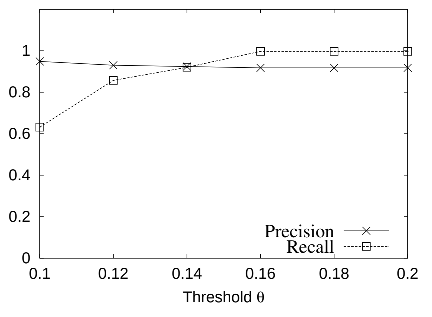
(a) TPCH-1



(b) TPCH-2



(c) IMDB



(d) DBLP

Figure 4.11: Accuracy results on TPC-H, IMDB and DBLP for varying thresholds θ.

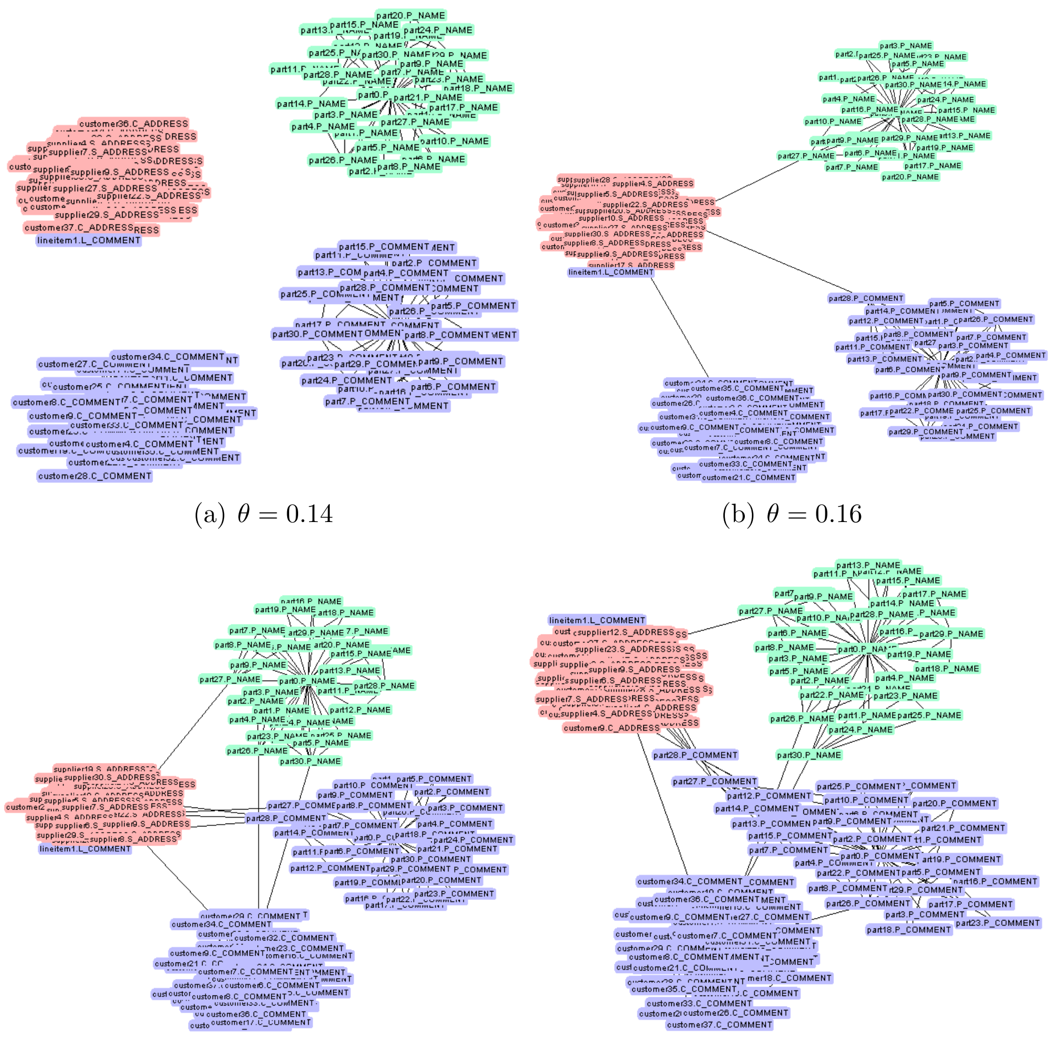
1	DISCOUNT	TAX	
2	SUPPKEY	AVAILQTY	
3	CUSTOMER.ACCTBAL	SUPPLIER.ACCTBAL	
4	CUSTOMER.ADDRESS	SUPPLIER.ADDRESS	LINEITEM.COMMENT

Table 4.6: Attributes that are incorrectly clustered together in TPC-H for $\theta = 0.12$.

in CUSTOMER.ADDRESS (due to the way addresses are generated in TPC-H), making these two columns indistinguishable; DISCOUNT contains all values in TAX; the same is true for SUPPKEY and AVAILQTY; 12.8% of values in SUPPLIER.ACCTBAL appear in CUSTOMER.ACCTBAL. Finally, LINEITEM.COMMENT has no intersection with ADDRESS, but the false clustering here occurs due to false positives produced by the Bloom filter in phase two.

Correlation clustering proves to be a robust way of separating attributes in the attribute graph. Figure 4.12 shows an attribute sub-graph of TPC-H, for varying θ from 0.14 to 0.2 (negative edges are removed for readability). For $\theta = 0.14$, the four clusters are totally disconnected (there are no positive edges between different attributes). Correlation clustering (or even connected components) in this case would separate the graph into four attributes. Our method is able to separate CUSTOMER.COMMENT views from PART.COMMENT views, while, for example, the method of comparing column names will fail in this case. On the other hand, CUSTOMER.ADDRESS and SUPPLIER.ADDRESS are clustered together, but this is clearly because 9997 out of 10000 addresses are the same. As θ increases, the number of positive edges across attributes increases as well. This is evident in Figure 4.12(d). However, after running correlation clustering we are still able to separate the graph into four attributes once again, with very few errors. For $\theta = 0.16$, the result is exactly the same as for $\theta = 0.14$. For $\theta = 0.18$, correlation clustering will place one view of PART.NAME in the attribute of CUSTOMER.COMMENT. For $\theta = 0.2$, one view of PART.COMMENT will be placed in the attribute for ADDRESS.

For IMDB we achieve 0.95 precision and 0.9 recall for θ ranging from 0.16 to 0.2. In our result, ACTOR.NAME and DIRECTOR.NAME are clustered together due to very large overlap of values. Since most directors are also actors, in this case choosing whether directors and actors should be treated differently depends on application context. In this case of course, a simple solution based on column names can provide an answer. Another problem is



(a) $\theta = 0.14$

(b) $\theta = 0.16$

(c) $\theta = 0.18$

(d) $\theta = 0.2$

Figure 4.12: An attribute sub-graph of TPC-H for varying thresholds θ.

column MEXICO.MOVIENAME which is not included in the movie names attribute. Some simple data analysis here shows that 14.0% of movie names in MEXICO.MOVIENAME start with *la/las/los* and 11.5% names start with *el*, making the distribution of this column significantly different from movie names in other views. Decreasing θ to 0.14 and 0.12 results in splitting SPAIN.MOVIENAME out as well, for the same reason. When using threshold $\theta = 0.1$, HONGKONG.MOVIENAME and TAIWAN.MOVIENAME are also separated. This is not surprising, since both mainly contain names in Chinese and have small overlap with the movie names in other views.

Finally, for the DBLP dataset we also achieve precision above 0.9 and recall above 0.997, for θ ranging from 0.16 to 0.2. The errors occur in four attributes. Here, AUTHOR.NAME and EDITOR.NAME are clustered together given that 596 out of 621 editors appear in the AUTHOR table as well. The same is true for ARTICLES.TITLE and INPROCEEDINGS.TITLE, since it seems that the majority of papers submitted to journal publications have the exact same title as the conference versions of the papers.

Overall, clearly it is in some cases difficult even for humans to decide what constitutes an attribute, without additional application dependent context. Our technique is able to separate major attributes very well, and make only minor mistakes that can either be corrected by supervision and simple statistical analysis, or by using orthogonal approaches (e.g., column name matching, if meaningful schema information exists).

4.6 Summary

We argued that discovering attributes in relational databases is an important step in better understanding and working with the data. Toward this goal, we proposed an efficient solution, based on statistical measures between pairs of columns, to identify such attributes given a database instance. Our solution was able to correctly identify attributes in real and synthetic databases with very high accuracy.

This work has been published as a full research paper in *2011 ACM SIGMOD/PODS Conference* [60].

CHAPTER 5

Join Query Discovery

In this chapter, we study the following problem: Given a database $\mathcal{D}$ with schema $\mathcal{G}$ and an output table *Out*, compute a join query Q that generates *Out* from $\mathcal{D}$. A simpler variant allows Q to return a superset of *Out*. This problem has numerous applications, both by itself, and as a building block for other problems arising in data mining, keyword search and schema mapping. Related prior work imposes conditions on the structure of Q which are not always consistent with the application, but are used for ease of computation. We discuss several natural SQL queries that do not satisfy these conditions and cannot be discovered by prior work.

We propose an efficient algorithm that discovers queries with arbitrary join graphs. A crucial insight is that any graph can be characterized by the combination of a simple structure, called a star, and a series of merge steps over the star. The merge steps define a lattice over graphs derived from the same star. This allows us to explore the set of candidate solutions in a principled way and quickly prune out a large number of infeasible graphs. We also design several optimizations that significantly reduce the running time. Finally, we conduct an extensive experimental study over a benchmark database and show that our approach is scalable and accurately discovers complex join queries.

5.1 Introduction

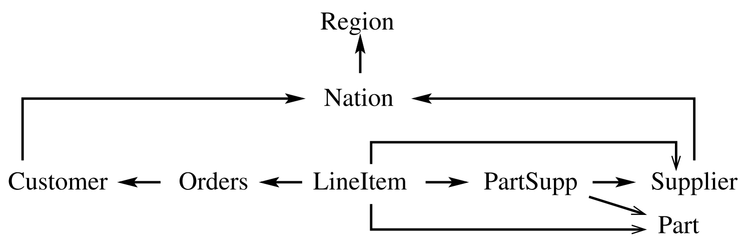
Database systems are adept at performing efficient computations expressed as SQL queries over large datasets, and much work in the literature has focused on improving the efficiency of answering SQL queries. In this chapter, we focus on the following inverse problem: Suppose that a user already has the answer to a SQL query, and her goal is to find the query itself. This scenario appears quite often in practice. Many database users compute a SQL answer into a spreadsheet or a view, then share it without annotating it with the generating query [56]. However, knowing the generating query can be very useful: e.g., someone may notice inconsistencies in the output and want to investigate, or they may want to generate a slightly different output for further analysis.

We discover join queries with *arbitrary graphs*, where each join in the generating query is an equi-join over foreign/primary key columns - a problem for which we are not aware of any prior solution. Partial results have been proposed before (see, e.g., [6, 14, 46, 52, 56]) for simple join graph structures such as trees, or subgraphs of the schema graph where each table appears at most once. While equi-joins over foreign/primary key columns are the most common kind of joins, many other joins are also used in practice, e.g., equi-joins between foreign keys, inequality joins over ordered attributes, etc. Our framework can be extended to support more join types if they are pre-defined in the schema; this may result in higher computation cost for some queries. Note, however, that computing a generating query with arbitrary arithmetic expressions in the joins is PSPACE-hard [56].

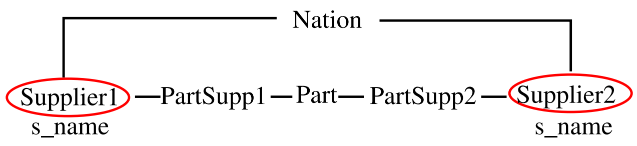
We now formalize the problem of join query discovery as follows:

Definition 5.1 (Join Query Discovery)**.** *Let $\mathcal{D}$ be a database with schema graph $\mathcal{G}$. Let Out be an output table generated from $\mathcal{D}$ by an unknown join query using the constraints in $\mathcal{G}$.* Join Query Discovery *is to compute a generating join query Q that produces table Out from the tables in $\mathcal{D}$, i.e., $out(Q) = Out$. In general, a complexity measure is defined over the queries, and the goal is to return the query with smallest complexity.*

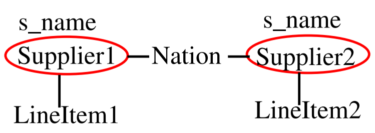
Because of their potential applications in other areas, we also consider a few variants of the problem. Thus, we may wish to compute a query that generates a superset of *Out*, or to compute multiple queries that generate (a superset of) *Out*, in order of increasing complexity. However, unless otherwise specified, our



(a) Schema graph of TPC-H.



(b) **RQ1**



(c) **RQ2**

Figure 5.1: The TPC-H schema and two **R**unning example **Q**ueries over it.

presentation assumes that we return the query whose output is exactly table Out.

Example: Figure 5.1(a) shows the schema graph of the TPC-H benchmark [55], which models a business environment and contains information about products, suppliers, customers, etc. Figures 5.1(b) and 5.1(c) show two queries (in graph form) over TPC-H; they serve as running examples throughout Chapter 5. Query RQ1 in Figure 5.1(b) outputs a table $Out1$ of pairs of supplier names (from table *Supplier*) that are located in the same country, and supply at least one identical product. To distinguish between two different instances of the same table, we add a counter after the name (e.g., *Supplier1* and *Supplier2*). The query corresponds to two cyclic traversals in the TPC-H schema: *Supplier-Nation-Supplier* and *Supplier-PartSupp-Part-PartSupp-Supplier*. However, the graph of RQ1 is not a subgraph of TPC-H, because we use multiple instances of *Supplier* and *PartSupp*. Query RQ2 in Figure 5.1(c) outputs a table $Out2$ of pairs of supplier names that are located in the same country and each supplies at least one line item. The two joins with the *LineItem* tables are a proper filter: deleting them changes the output. The red ovals indicate the tables from which the projection columns are selected. Tables $Out1$ and $Out2$ are overlapping but not identical, so we must distinguish between RQ1 and RQ2 in our answers.

Any algorithm that solves this problem must exhibit *scalability*. Even for a simple schema like TPC-H, the number of candidate graphs is usually super-exponential in the graph size, and many graphs behave like cross-products (because they contain multiple copies of the same table, and the size of the query output can be at least quadratic in the size of such a table). Thus, while one could design various brute-force approaches that enumerate and test all candidate queries (up to some size), such approaches will not scale in practice.

We also emphasize *correctness*, i.e., we are able to discover a query that generates the exact table *Out* (not a subset or superset of it). Below we show why related work falls short of this goal. Finally, we discuss *completeness*: our algorithm can discover *all* query graphs (up to some fixed complexity) that generate exactly *Out*.

In summary, we propose an efficient method for reverse engineering arbitrary join queries. Our contributions in this chapter can be summarized as follows:

- We prove that any graph can be characterized by the combination of a simple structure, called a *star*, and a series of merge steps over the star. The merge steps define a *lattice* over graphs derived from the same star. The proof relies on "unwinding" the Euler tour of a graph.
- We propose a novel algorithm that uses the star and lattice constructs to reverse engineer arbitrary join queries.
- We design several optimizations that significantly reduce the running time of the algorithm, making it practical for large, complex queries.
- We conduct an extensive experimental study over the TPC-H benchmark and for complex query graphs. The study shows that our algorithm, enhanced with our optimization strategies, is fast and scalable. We expect this good performance to extend to other databases, as well.

5.2 Preliminaries

5.2.1 Overview

We start by identifying possible sets of projection tables. For each set there is a combinatorial explosion of candidate queries (as we see below). Thus, the ability to quickly validate or eliminate candidates is crucial for the algorithm.

Challenges: Before giving an intuition for our ideas, we illustrate the main challenges of the problem. We show where prior methods fail, and why naive

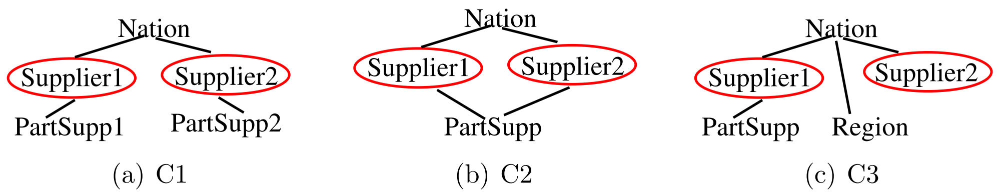


Figure 5.2: Example candidate graphs for RQ2: naive approach.

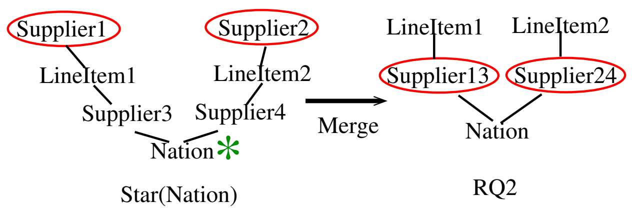
approaches cannot scale. We then use this discussion to motivate the techniques we propose.

Suppose we start with table $Out = out(RQ2)$, where RQ2 is the query in Figure 5.1(c), and that we correctly identify the projection tables to be two distinct copies of *Supplier*. The obvious next step is to construct a graph that connects these copies via join edges. For this simple case, prior approaches would return connecting paths of some maximum length ℓ_p - let's call them "core paths"; e.g., for $\ell_p = 2$, some core paths are *Supplier1-Nation-Supplier2*, *Supplier1-PartSupp-Supplier2*, and so on. However, this is not enough: to discover query RQ2 we need to extend prior approaches and add the two "hanging" edges *LineItem1-Supplier1* and *LineItem2-Supplier2* to the core path *Supplier1-Nation-Supplier2*. How do we explore all possibilities in a principled way?

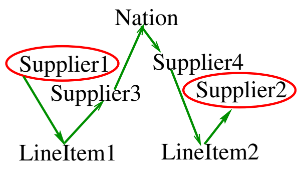
A naive approach is to fix some number of additional edges (say, 2), and enumerate all ways for adding them to the core path. Figure 5.2 shows three such candidate graphs, out of more than 30 possible graphs (over all core paths of length 2).

The crucial issue to keep in mind is that testing all potential candidates (by running each query against the database and comparing its output to *Out*) can be extremely expensive. For RQ2, potential candidates (and indeed, the query itself) behave almost like cross-products of table *Supplier* with itself. In general, the number of candidates increases super-exponentially in the graph size, and testing them may take hours or even days of computation; e.g., starting from the output of query Q5 in Table 5.3 (Section 5.5), there are so many candidate queries, even after significant pruning by our algorithm, that the testing could not finish after a day. However, after applying several optimization ideas, our approach managed to correctly discover query Q5 in 3 minutes.

A basic observation that we exploit below is that some candidate graphs need not be tested. Consider graphs $C1$ and $C2$ in Figure 5.2. Clearly,



(a) RQ2 is obtained from $Star(Nation)$ via table merges.



(b) Touring RQ2 to get $Star(Nation)$.

Figure 5.3: Illustration of our graph characterization result.

$out(C2) \subseteq out(C1)$. If $Out \not\subseteq out(C1)$, we can rule out $C2$ without testing. Conversely, if $Out \subset out(C2)$ (strict inclusion) and we want a solution that matches Out exactly, then we rule out $C1$. However, no relationship can be inferred between, e.g., $out(C3)$ and $out(C2)$, so we can prune neither based on the other. Hence, we need a formal way of reasoning about candidate graphs.

Our techniques We now give a brief description of how we solve the challenges discussed above. The exact details are in the next two sections.

Our main insight is as follows: Any query graph can be generalized into a union of disjoint paths connecting its projection tables to a center table. We refer to this union as a star, and the center table as the *star center*. Figure 5.3(a) illustrates this for the graph RQ2: The graph $Star(Nation)$ connects the star center *Nation* to projection tables *Supplier1* and *Supplier2* via disjoint paths. We show that $out(RQ2) \subseteq out(Star(Nation))$, which is why the star is a generalization. To specify RQ2, we add two restrictions to the star: tables *Supplier1* and *Supplier3* are the same, and tables *Supplier2* and *Supplier4* are the same. If we merge *Supplier1* and *Supplier3* into *Supplier13* (the naming convention shows which copies are collapsed), and similarly *Supplier2* and *Supplier4* into *Supplier24*, we obtain a graph isomorphic to RQ2. Thus, a complete description of $RQ2$ consists of $Star(Nation)$ and the sets of tables to be merged.

How did we come up with $Star(Nation)$? Imagine we take a tour around RQ2 as follows: start in *Supplier1*, go to *LineItem1*, then back to *Supplier1*, then *Nation* etc. The tour ends in *Supplier2*. We “unwind” this tour by making a new copy of a table each time we revisit it: e.g., when coming back from *LineItem1* to *Supplier1*, we rename it *Supplier3* (new copy); see Figure 5.3(b). The unwound tour is $Star(Nation)$. To obtain RQ2, we merge back copies of the same table. Lemma 5.1 shows that such a construction exists for any graph.

There are two advantages to using this construction. First, we obtain a clean and simple algorithm for enumerating candidate solutions: We discover stars that are potential generalizations of the solution, and explore possible table merges in each star. For the latter step, we use a *lattice* structure rooted at that star, that describes how graphs are derived from each other via merge steps. Second, lattices allow us to avoid testing a significant number of potential candidates; see Section 5.3.4.

Finally, we design several optimizations so we can discover complex solutions faster. We illustrate this on the graph RQ1 from Figure 5.1(b). It has an unwound tour of length 10. The resulting star $star(Nation)$ has radius 5 (i.e., the maximum distance from the center to a projection table). Exploring all stars up to radius 5 generates too many candidates which are expensive to test. Instead, we observe that RQ1 is the *intersection* between two smaller graphs: the upper chain (a star of radius 1 centered at *Nation*), and the lower chain (a star of radius 2 centered at *Part*). These graphs are discovered in early iterations, and can be combined into a solution. We describe the general form of this optimization in Section 5.4, along with several other ideas. The intersection approach reduced the running time from a day to 3 minutes for the example mentioned above. Full details are in Section 5.5.

5.2.2 Definitions

Let $\mathcal{D}$ be a database with tables $\mathcal{T} = \{R_1, \ldots, R_n\}$, which we refer to as *base tables*. Let $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ denote the schema graph of $\mathcal{D}$, defined in the usual way: the nodes correspond to the tables in $\mathcal{T}$ and for each fk/pk constraint between (single- or multi-) columns of R_1 and R_2, there is an edge in $\mathcal{E}$ between the nodes corresponding to R_1 and R_2. Each edge is labeled by its respective constraint. Thus, $\mathcal{G}$ may contain self-loops, as well as parallel edges between the same pair of tables. We consider $\mathcal{G}$ undirected. However, we sometimes depict the edges directed from fk to pk, to convey more information.

Definition 5.2. *A* query graph *is an arbitrary graph $Q = (V, F)$ with the following properties:*

(a) Q is compatible *with the schema graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, i.e., there exists a labeling function $\lambda : \{V \cup F\} \rightarrow \{\mathcal{T} \cup \mathcal{E}\}$ such that for each edge $(A, B) \in F$,*

$\lambda((A, B))$ is a fk/pk constraint in $\mathcal{G}$ between $\lambda(A)$ and $\lambda(B)$.

(b) Q has a subset of specially marked vertices $\mathcal{P} \subset V$ called projection nodes (or tables), $1 \leq |\mathcal{P}| \leq |V|$. Each $A \in \mathcal{P}$ is associated with one or more columns from $\lambda(A)$: these are the projection attributes in the SQL query corresponding to graph Q, i.e., the attributes in the SELECT ... FROM *portion of the query.*

The complexity *of Q is $|V|$, i.e., the number of nodes (tables) in the query graph.*

Other definitions for the complexity of Q are also possible, e.g. $|F|$ (the number of joins in Q). No complexity measure is obviously optimal, so we choose $|V|$ because it is simple and intuitive. By abuse of notation, we refer to a join query and its query graph interchangeably.

Example The graph in Figure 5.1(b) is compatible with TPC-H: $\lambda(Part) = Part$, $\lambda(Supplier1) = \lambda(Supplier2) = Supplier$, etc. Also, $\lambda((Supplier1, Nation)) = \lambda((Supplier2, Nation)) = Supplier.s\_nationkey\text{–}Nation.n\_nationkey$, etc. (To improve readability, we omit the edge labels from all figures. Since the TPC-H schema does not have parallel edges, this is unambiguous.) The projection nodes in Figure 5.1(b) are *Supplier*1 and *Supplier*2, marked by a red oval; each has one projection attribute, *s_name*. We omit projection attributes in subsequent figures, for simplicity. The graph complexity is 6.

Definition 5.3. *A* star *is a union of paths which are mutually disjoint except for a common endpoint, called the* star center*. Its* radius *is the length of the longest path from the star center to an endpoint. The star is compatible with a schema graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ if its nodes and edges can be labeled as in Definition 5.2(a).*

The star generalizes *a query graph Q if there is a one-to-one correspondence between the projection tables of Q and the star endpoints, and a one-to-many correspondence between the edges of Q and the star edges.*

As an example, recall that $Star(Nation)$ in Figure 5.3(a) generalizes the graph RQ2. A graph can be generalized by many stars of different radii. By abuse of notation, a tuple-level instantiation of a star will also be called a star, with a corresponding (tuple-level) star center. Our meaning will be clear from the context, but we will be more precise when necessary.

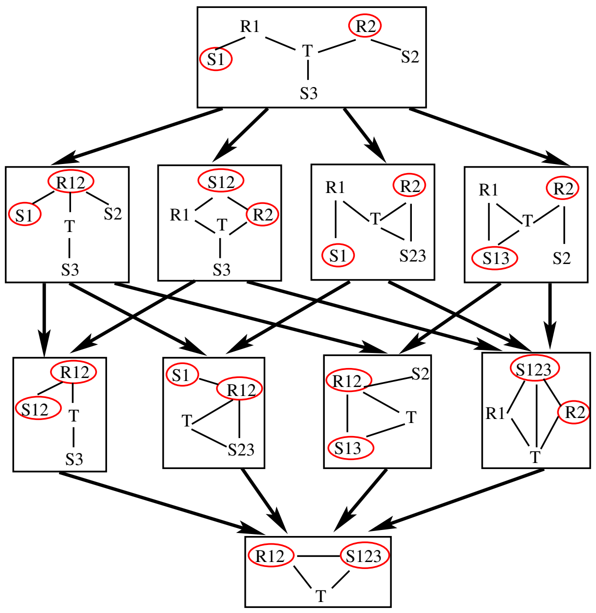


Figure 5.4: Example of lattice. An edge corresponds to a merge step.

Definition 5.4. *Let $Q = (V, F)$ be a query graph compatible with a schema graph (via function λ), and with projection nodes $\mathcal{P}$. A* merge step *in Q involves a pair of nodes $R1, R2 \in V$ such that $\lambda(R1) = \lambda(R2)$ and proceeds as follows:*

- replace $R1$ and $R2$ by a new node $R12$: $\lambda(R12) = \lambda(R1)$;

- let the edges incident on $R12$ be the union *of the edges previously adjacent on $R1$ or $R2$ (eliminating duplicates);*

- if neither $R1$ nor $R2$ are projection nodes in $\mathcal{P}$, then $R12 \notin \mathcal{P}$. Else, $R12 \in \mathcal{P}$; its set of projection attributes is the union of projection attributes in $R1$ and $R2$ (if any).

The resulting graph is denoted $Q_{R1=R2}$.

Definition 5.5. *Let Q be a query graph. The lattice structure rooted at Q, $Lattice(Q)$, is defined recursively as follows:*

- $Lattice(Q)$ contains Q;

- For any graph $G_1 \in Lattice(Q)$ and any graph G_2 derived from G_1 via a merge step, $Lattice(Q)$ contains G_2, as well as a directed edge $G_1 \to G_2$.

Example In Figure 5.4, let Q be the topmost graph, i.e., the lattice root. Then the graphs on the row below Q are, respectively, $Q_{R1=R2}$, $Q_{S1=S2}$, $Q_{S2=S3}$ and $Q_{S1=S3}$. In the next row, the leftmost graph is equal to both $(Q_{R1=R2})_{S1=S2}$ and $(Q_{S1=S2})_{R1=R2}$. Hence, it has two parent nodes, $Q_{R1=R2}$ and $Q_{S1=S2}$.

The following lemma is proven in Section 5.3.

Lemma 5.1. *Let Q be a query graph. There exists at least one star $\mathcal{S}$ that generalizes Q, and Q is a node in the latttice rooted at $\mathcal{S}$.*

5.3 Query Generation

We now describe our algorithm for computing candidate query graphs, given an output table Out and a database $\mathcal{D}$ with schema graph $\mathcal{G}$. Consistent with prior work, we assume that all joins are fk/pk. This is for ease of presentation only: any fk/fk join is equivalent to two consecutive joins fk/pk, pk/fk involving their primary key.

Algorithm 5.1: Overview of our approach

```
Input: $\mathcal{D}, \mathcal{G}$: DB and its schema, $Out$: output table
Output: $Q$: generating query of $Out$
1  foreach $depth\ d \leq d_{max}$ do
2      Step 1:
3      $(\{Trees(Out.A_i)\}, StarCtrs) := ExploreSchema(d)$;
4      $\Theta \subseteq Out$: random subset of tuples
5      foreach $\theta \in \Theta$ do
6          foreach $\mathbf{T} \in \cup_i Trees(Out.A_i)$ do
7              Step 2.a: $\mathbf{T} := ExploreInstanceTree(\mathbf{T}, \theta)$;
8          Step 2.b:
9          $StarCtrs := UpdateStarCenters(StarCtrs, \theta)$;
10     foreach $C \in StarCtrs$ do
11         Step 3: $CandGraphs := ExploreLattice(C)$;
12     Step 4: $SolutionSet := Test(CandGraphs)$;
13     if $SolutionSet \neq \emptyset$ then
14         output $SolutionSet$;
15         exit;
```

Our overall approach is described in Algorithm 5.1. We enumerate all s-tars that might generalize the solution (Step 1), explore their lattices (Step 3), then test the candidate graphs in those lattices (Step 4). However, to achieve scalability, we interpose partial validations in Steps 2 and 3: We select a random subset of output tuples $\Theta \subseteq Out$ and eliminate those graphs that cannot generate all tuples in Θ. In our experiments, a small Θ (of size at most 5) was

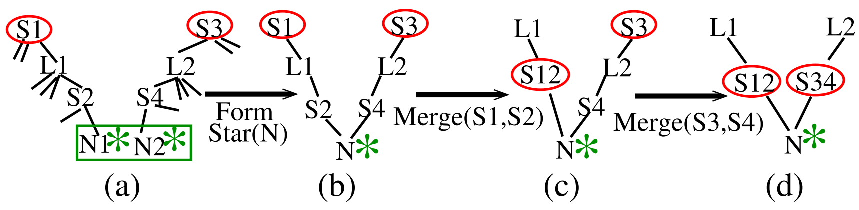


Figure 5.5: Computing the query in Figure 5.1(c) via Algorithm 5.1.

sufficient to prune out most invalid graphs, and the validation overhead was negligible.

Figure 5.5 illustrates how our algorithm discovers query RQ2 from Figure 5.1(c). We abbreviated table names in the obvious way and added a unique count to distinguish nodes labeled by the same table; e.g., nodes $S1, S2, S3$ and $S4$ are different instances of table *Supplier*, $L1, L2$ are instances of *LineItem*, and so on.

In Step 1 we generate all stars of radius d that could generalize the solution graph. First, we determine a candidate set of projection tables. We then work backwards from the projection tables towards the star center. This means exploring outward from each projection table along join edges to generate a tree of depth d. Figure 5.5(a) shows two trees rooted at S and of depth $d = 3$. Any table that occurs in all trees is a star center. For example, in Figure 5.5(a), tables S, N and L are all star centers, and each leads to a different star. Moreover, since table S occurs multiple times per tree, we also specify which occurence is the star center (and enumerate all choices). Figure 5.5(b) depicts the star centered at N. This process is similar to the “distinct root semantics” from keyword search; see, e.g., [31, 47] and references therein. We execute it first at schema level (Step 1), then at instance level (Step 2).

After forming a star, we explore its lattice. Figures 5.5(c) and (d) show how two merge steps in the lattice of *Star*(N) generate a graph isomorphic to the desired solution RQ2. Lattices are explored in Step 3. As far as we are aware, this is a novel approach which leads to a principled exploration of the query space. Moreover, it allows us to quickly prune out a large number of candidate queries, as well as design more efficient testing strategies in Step 4.

We detail each of the four steps below. For scalability reasons, we execute the above framework in parallel, on several levels: over all choices of projection tables, all choices of star centers, and all instantiations from a tuple in Θ. This requires maintaining several data structures, as we illustrate via an example.

5.3.1 Step 1: Schema Exploration and Pruning

This step, denoted $ExploreSchema(d)$ in Algorithm 5.1, is executed primarily at schema level (with one exception). It consists of the following phases:

Candidate projection tables: Compute, for each column $Out.A$, the set $Cand(Out.A)$ of all columns in the base tables that contain $Out.A$: these are all the possible columns from which $Out.A$ could have been generated via projection. See Figure 5.7(a) for an example. The candidate projection tables are tables whose columns appear in the union of lists $Cand(Out.A)$.

This is the only computation in Step 1 that depends on the size of table Out: the more distinct values a column $Out.A$ contains, the more expensive it is to check that it is included in a base table column. On the other hand, the larger the number of distinct values in $Out.A$, the smaller the expected size of $Cand(Out.A)$, which in turn can lead to significantly fewer trees and stars.

Schema-level Trees: These data structures guide our instance-level exploration in Step 2. For each candidate projection column $R.C$, we compute a tree $\mathcal{T}(R.C)$ rooted at R and of depth d. Its nodes are base tables, and its edges correspond to a fk/pk constraint between them. To distinguish among multiple nodes labeled by the same table, we add a count after the table name. Tree $\mathcal{T}(R.C)$ is computed via a bfs exploration of the schema, starting from R. Figure 5.7(b) shows three trees of depth 2 over TPC-H: $\mathcal{T}(PS.suppcost)$, $\mathcal{T}(L.shipdate)$ and $\mathcal{T}(O.orderdate)$. Note that a schema edge may appear multiple times on a path; e.g., the edge *PS.ps_partkey–P.p_partkey* appears twice on the path *PS1-P1-PS3*. If there are multiple fk/pk constraints between two tables S and T, then multiple edges $S\text{i}-T\text{j}, \ldots, S\text{i}-T\text{k}$ are created. We also create edges $S\text{i}-S\text{j}$ for each self-join constraint of S.

For each column $Out.A_i$, let $Trees(Out.A_i)$ be the set of all trees $\mathcal{T}(R.C)$, where $R.C \in Cand(Out.A_i)$. Hence, $Trees(Out.B) = \{\mathcal{T}(L.rcpdate), \mathcal{T}(L.shipdate)\}$. Although the two trees are isomorphic, they will be explored differently at instance level.

Schema-level Star Centers: We compute a bit vector $StarCtrs$ which is 1 for tables that are star centers, and 0 otherwise. Details are omitted for lack of space. We mark all tree nodes labeled by a star center and call them

tid	pkey	skey	suppcost
1	a	α	20
2	a	β	20
3	a	γ	10
4	b	β	70
5	d	γ	20

(a) *PS*

tid	okey	orderdate	totprice
18	o_1	7/22/12	20
19	o_2	7/25/12	10
20	o_3	7/23/12	70

(b) *O*

tid	skey
10	α
11	β
12	γ
13	δ

(c) *S*

tid	pkey	skey	okey	rcptdate	shipdate	extprice
14	a	α	o_1	7/29/12	7/24/12	20
15	a	β	o_2	7/28/12	7/28/12	20
16	a	γ	o_3	7/30/12	7/24/12	10
17	b	β	o_3	7/24/12	7/24/12	70

(d) *L*

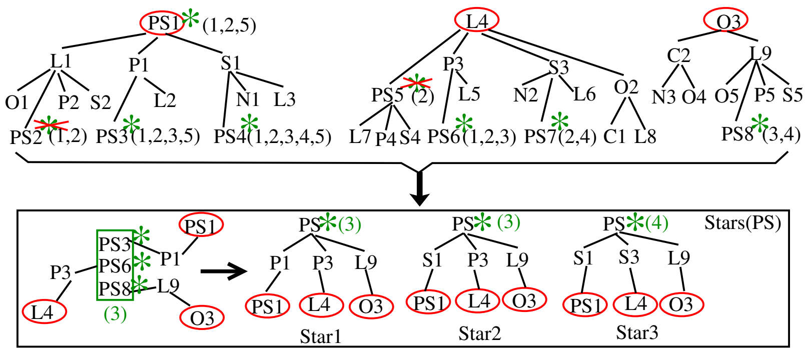
tid	pkey	retailprice
6	a	20
7	b	70
8	c	10
9	d	30

(e) *P*

Figure 5.6: TPC-H instance (only relevant tables and columns are shown; column names are abbreviated).

Out.A	*Out.B*	*Out.C*
O.totprice	L.rcptdate	O.orderdate
L.extprice	L.shipdate	
P.retailprice		
PS.suppcost		

(a) *Cand* lists for output columns.



(b) Trees/stars instantiated by $\theta = (20, 7/28/12, 7/23/12)$.

Figure 5.7: Algorithmic steps for table *Out* = "SELECT PS.suppcost, L.shipdate, O.orderdate FROM PartSupplier as PS, PartSupplier as PS1, Part as P, Supplier as S, LineItem as L, LineItem as L1, Orders as O WHERE PS1.skey = S.skey and S.skey = PS.skey and PS1.pkey = P.pkey and P.pkey = L.pkey and PS1.pkey = L1.pkey and PS1.skey = L1.skey and L1.okey = O.okey" (its graph is isomorphic to *Star*2).

starred nodes; see, e.g., the PS nodes in Figure 5.7(b) (star marks for other nodes were omitted for clarity). When a set of starred nodes from different trees can be merged to form a star center, we call the set a star center, as well (slightly overloading the term).

Bottom-up Pruning: Stars are unions of paths between tree roots (i.e., projection tables) and starred nodes labeled by the same table (i.e., a star center). Hence, we can delete the subtrees below these starred nodes, as long as they contain no other starred node. Equivalently, we prune each tree bottom-up until all its leaves are starred.

5.3.2 Step 2: Instance Trees and Star Centers

In this phase we explore at instance level the trees computed in Step 1, starting from a set of randomly selected output tuples $\Theta \subseteq Out$. We assume that all tuples in the database have a unique identifier *tid*; e.g., this can be their table name followed by their primary key value, or a unique numeric value assigned in a pre-processing phase. During this step some of the starred nodes become un-marked. Whenever we refer to starred nodes, we mean those nodes that are currently marked.

We first illustrate our ideas over the example in Figure 5.7, starting from output tuple $\theta = (20, 7/28/12, 7/23/12)$. The algorithm iterates over all trees; we focus on the leftmost one, $\mathcal{T}(PS.suppcost)$. First, we annotate the root with the list of tid's from table PS that have value 20 (=$\theta[1]$) on column *suppcost*; this is because $PS.suppcost$ is a candidate for the first output column $Out.A$. The result is a list $TID(PS1, \theta) = (1, 2, 5)$, since tuples 1, 2, 5 have $PS.suppcost = 20$. We then traverse down the tree and recursively annotate each node, as follows. After traversing edge $PS1 - L1$ (labeled by $(PS.pkey, PS.skey) = (L.pkey, L.skey)$), we create $TID(L1, \theta)$ to contain all tuples in $L1$ that join with any tuple in $TID(PS1, \theta)$ over this edge. Thus, $TID(L1, \theta) = (14, 15)$ since tuples 14 and 15 join with tuples 1 and 2, respectively. Similarly, $TID(PS2, \theta) = (1, 2)$ contains all tuples in PS that join with tuples 14 or 15 via the same constraint (which also labels the edge $L1 - PS2$). We annotate all nodes this way. For clarity, Figure 5.7(b) only shows the TID lists of PS nodes.

Intersecting TID lists yields star centers whose respective stars generate tuple θ. For example, since $tid = 3$ is in the intersection of TID lists for nodes $PS3$, $PS6$ and $PS8$, the star obtained by merging these nodes generates θ at instance level (see $Star1$ in Figure 5.7(b)). We say that $tid = 3$ *validates* the starred set $\{PS3, PS6, PS8\}$. On the other hand, $tid = 1$ and $tid = 2$ do not appear in the third tree. This means that: (a) no instance-level star centered at either tuple 1 or tuple 2 can generate θ, so we call these tuples *invalid*; and (b) the nodes $PS2$ and $PS5$, which only contain invalid tuples, cannot form a star center.

While there are $4 \times 3 \times 1 = 12$ different sets of starred PS nodes (one from each tree) that we could merge to form a star center, we do not need to enumerate all of them. Instead, since only tuples 3 and 4 occur in the third tree, it means that any valid set contains either nodes in which 3 is stored, or nodes in which 4 is stored. Hence, there are $2 \times 1 \times 1 + 1 \times 1 \times 1 = 3$ valid sets. The resulting three stars are depicted in Figure 5.7(b).

Inverted lists: To efficiently determine valid tuples and sets, we compute, for each tid in a starred node, the list $IL(tid, \theta)$ of nodes in which tid occurs. In Figure 5.7(b), $IL(3, \theta) = \{PS3, PS4, PS6, PS8\}$. This implicitly represents the two validated starred sets $(A, PS6, PS8)$ for $A \in \{PS3, PS4\}$. The set of all validated starred sets, denoted $Valid(PS, \theta)$, contains all inclusion-maximal IL lists and encodes all stars of center PS that generate θ. In Figure 5.7(b), $Valid(PS, \theta) = \{IL(3, \theta), IL(4, \theta)\}$.

In Algorithm 5.1, procedure $UpdateStarCenters(StarCtrs, \theta)$ maintains valid starred sets via this approach.

Pruning and negative witnesses: Pruning occurs both in Step 2.a and in Step 2.b, and results in changes to the data structures (denoted by the assignment operator in Algorithm 5.1). During procedure $ExploreInstanceTree(\mathcal{T}, \theta)$, if a list $TID(S\text{j}, \theta)$ is empty, we delete node Sj (and its subtree) from $\mathcal{T}$. We say that θ is a *negative witness* for the edge e between Sj and its parent, since θ cannot be generated by any query that contains e. We then apply bottom-up pruning (see Section 5.3.1) until all leaves of $\mathcal{T}$ are starred. Note that the deleted nodes will not be explored in any subsequent calls to $ExploreInstanceTree(\mathcal{T}, \theta')$ for other $\theta' \in \Theta$ (at current depth d).

Similarly, during $UpdateStarCenters(StarCtrs, \theta)$, θ can be a negative witness for certain starred nodes. Let Si be a starred node such that $TID(S\text{i}, \theta)$ contains only invalid tid's; hence, Si cannot form a star center. We delete its star mark, then apply bottom-up pruning until all leaves are starred. For example, in Figure 5.7(b), θ is a negative witness for $PS2$ and $PS5$: their star marks are removed and $PS2$ will be deleted by bottom-up pruning.

Cross-tuple pruning: Let $\theta, \theta' \in \Theta$ be two random output tuples. In general, the lists in $Valid(S, \theta)$ and $Valid(S, \theta')$ are different. A star must generate both θ and θ', so its star center must be validated by both. Let $Valid(S)$ be the list of star centers validated by all $\theta \in \Theta$. We maintain $Valid(S)$ incrementally, as a cross-intersection with each new $Valid(S, \theta)$: For each list $Ł \in Valid(S)$ and each inverted list $IL(tid, \theta') \in Valid(S, \theta')$, we compute $Ł' = Ł \cap IL(tid, \theta')$. If $Ł'$ contains at least one table from each set $Trees(Out.A_i)$, $1 \leq i \leq k$, then $Ł'$ is a valid list. The new set $Valid(S)$ contains all valid lists $Ł'$. If a starred node Si does not occur in any list of $Valid(S)$, we remove its star mark then apply bottom-up pruning. If $Valid(S) = \emptyset$, then all S nodes are un-starred, and we remove S from $StarCtrs$.

The more random tuples θ we explore, the higher the chance of finding negative witnesses and reducing the tree sizes and the number of validated starred sets. However, this comes at the cost of increasing the running time of Step 2. We explore the tradeoffs in Section 5.5.

5.3.3 Step 3: Exploring Lattices

Let S be a star center table. For each starred set $\{S_1, \ldots, S_k\}$ represented in $Valid(S)$ (where S_i is a node in $Trees(Out.A_i)$), we form a star as a union of paths between each S_i and its tree root; the nodes $S_1, \ldots, S_k$ are merged into a single node S. Figure 5.7(b) illustrates this process for $Star1$ and shows all three stars centered at PS (we assume that $Valid(S) = \{IL(3, \theta), IL(4, \theta)\}$).

We then build the lattices rooted at these stars. This requires determining sets of *mergeable* tables, by maintaining lists $Merge(R)$ similar to the lists $Valid(S)$. For lack of space, we omit the details. An important property of the lattice is that we can compute any candidate solution in it without materializing its ancestors (by using the $Merge(\cdot)$ lists). This allows us to use different testing strategies in Step 4.

5.3.4 Step 4: Query Testing

The lattice structures imply two important relationships between candidate graphs from the same lattice. If graph Q_1 is the ancestor of graph Q_2 in a lattice, then: 1. Q_2 has smaller complexity than Q_1 (each merge step reduces the node complexity by 1); and 2. The output of Q_2 is included in the output of Q_1 (Q_2 can be regarded as Q_1 with additional conditions that force multiple table copies to be identical).

The first relationship suggests a strategy of testing the lowest graphs in a lattice first: if, e.g, Q_2 passes the test, there is no need to test Q_1 since it has higher complexity. On the other hand, the second relationship suggests the opposite strategy of testing the highest graphs first. If the output of $Q1$ is not a superset of Out, there is no point in testing any of its descendants. In particular, if the root of a lattice does not generate a superset of Out, we can drop all the lattice nodes from the testing phase. However, testing a lattice root tends to be expensive, since the star contains a large number of tables and it may behave like a cross-product query.

We test both strategies in Section 5.5. Intermediate strategies can also be considered, such as testing graphs in the middle of the lattice, or testing only a subgraph of a query graph (if the subgraph is invalid, then all graphs in the lattice that contain that subgraph are invalid). Since the same graph may appear in multiple lattices, we store the graphs tested so far. Before a candidate solution Q is tested, we check for isomorphism with the stored graphs. Due to our node labeling procedure, this can be done in time linear in the number of edges in Q.

Problem Variants: As discussed in Section 5.1, some variants of our problem may accept a solution Q that generates a superset of Out. We handle this by modifying Step 4 accordingly: a graph passes the test if it generates a superset of Out (instead of an exact match). In general, this leads to much faster executions of our algorithm, since a solution is usually found at a small depth d.

Other variants may ask for all solutions Q (up to a certain depth d_{max}), for either the exact match or the superset semantics. In that case, we simply delete the "**exit**" statement from Algorithm 5.1.

Correctness and completeness: Any graph Q returned by Algorithm 5.1 passed the test $out(Q) = Out$ in Step 4; thus, Algorithm 5.1 is always correct.

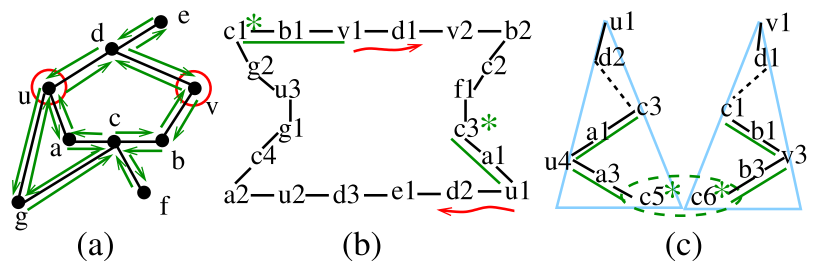


Figure 5.8: Proof of Theorem 5.1: (a) A query graph Q (black edges) and its directed version Q_d (green edges); (b) Modified Euler tour E_m; (c) Discovering a star whose lattice contains Q.

Assuming d_{max} is "large enough", our algorithm is also complete if it does not contain the "**exit**" clause (with the '**exit**" clause present, some queries may never be returned, because they have simpler equivalent queries). We now formally state this claim as a theorem below.

Theorem 5.1. *Let Q be an arbitrary query graph that generates table Out over a database $\mathcal{D}$ with schema $\mathcal{G}$. Then Algorithm 5.1 (with the "***exit***" statement deleted) outputs Q when provided with the input $(\mathcal{D}, \mathcal{G}, Out)$ (for d_{max} sufficiently large).*

Proof. To prove Theorem 5.1, we first prove Lemma 5.1.

Proof of Lemma 5.1 We illustrate the main ideas in Figure 5.8. In the following, we use small letters (e.g., x, y) as unique identifiers of graph nodes. These are distinct form the node labels (e.g., $\lambda(x) = \lambda(y) = R$), which denote tables and may be the same for many nodes. The proof has two phases: In the first, we create copies $x1, x2, \ldots$ of a node x, with $\lambda(x\mathrm{i}) = \lambda(x)$. In the second, we merge back all copies $x1, x2, \ldots$ of the same node x. It is important to note that the merge does not involve any nodes yi that may have the same table label, but are copies of a different node y.

Let Q_d be the directed graph obtained from Q by replacing each undirected edge $x - y$ by a pair of directed edges $x \to y$ and $y \to x$. It is well known that Q_d admits at least one Euler tour E_d.[1] We travel along E_d starting from an arbitrary position and enumerate the nodes in the order they are encountered, incrementing a counter for each node every time the node is visited. Figure 5.8(b) illustrates such a trip around an Euler tour of the (directed) graph in Figure 5.8(a); the trip starts in node $c1$ and visits $b1$ first. This yields

[1]An Euler tour traverses each edge once, but may visit a node multiple times.

a modified tour, denoted E_m, with the same number of edges as E_d but with more nodes; each node has degree 2 in E_m. We make all edges in E_m undirected.

Let c be an arbitrary node in the original graph Q. We show that Algorithm 5.1 discovers a star centered at c which contains Q in its lattice, and the proof follows. Let u be a projection node in Q. We define the following trip: starting from $u1$, travel once around E_m, ending back in $u1$; then reverse direction and travel back to the nearest copy of c. Figure 5.8(b) illustrates two such trips, from $u1$ and $v1$. The red arrows show the initial directions of each trip, and the green lines show the portion of the tour that is visited twice (once in each direction).

The trip from $u1$ visits some nodes multiple times. Replace those nodes by new copies, incrementing their counters appropriately. The result is a path $\Pi(u, c)$. Figure 5.8(c) shows $\Pi(u, c)$ and $\Pi(v, c)$; e.g., in $\Pi(u, c)$, the tour portion *u1-a1-c3* that was traversed in reverse direction is replaced by *u4-a3-c5*, which contains new nodes not in E_m. Form $\mathit{Star}(c)$ by connecting the paths at c. Then $\mathit{Star}(c)$ generalizes Q. In the lattice of $\mathit{Star}(c)$, merge all copies $x1, x2, \ldots$ of a node x, for all x. The result is graph Q.

Proof of Theorem 5.1 We use the notations from the above proof. Let $d = \max_u |\Pi(u, c)|$ be the radius of $\mathit{Star}(c)$. Then Algorithm 5.1 discovers $\mathit{Star}(c)$ in iteration d. It remains to prove that it does not discard Q because of pruning. Since Q generates Out, for any $\theta \in \Theta \subseteq \mathit{Out}$ and for any node x in Q, there exists a tuple $t(x, \theta)$ such that the tuple set $\{t(x, \theta)\}$ satisfies Q and generates θ via projections. This implies that $\{t(x, \theta)\}$ satisfies all paths $\Pi(u, c)$ (with $t(x, \theta)$ instantiating all copies $x1, x2 \ldots$ of node x). Therefore, Step 2.a of our algorithm does not prune any edge on these paths; and Step 2.b maintains, for each node x, the list $\{x1, x2 \ldots\}$ of all its copies in the union of paths $\Pi(u, c)$ as mergeable. Hence, Q is a candidate solution in Step 3, and it passes the test in Step 4.

□

5.4 Optimizations

As we show in Section 5.5, the running time of Algorithm 5.1 is dominated by Steps 2 and 4. For most queries, Step 4 is the bottleneck. Its running time increases significantly when going from depth d to $d + 1$: the number of

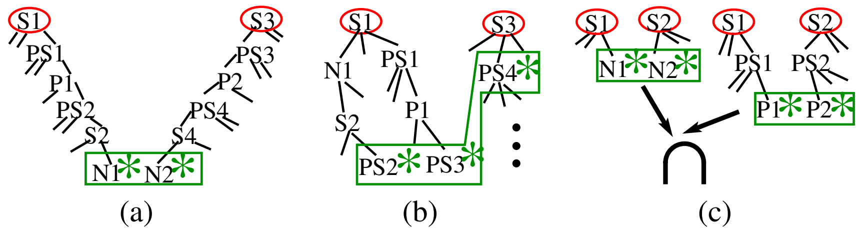


Figure 5.9: Computing the query in Figure 5.1(b) via Algorithm 5.1: (a)no optimizations, $d = 5$; (b) generalized stars, $d = 3$; (c) intersection, $d = 2$.

candidate solutions can increase (super)geometrically with each outer iteration, and the candidate graphs are larger, making their testing more expensive. The running time of Step 2 also depends on d, as well as the size of the TID lists in the tree nodes. Below, we describe several optimizations that address both issues.

5.4.1 Decreasing the depth d

We describe two optimizations that decrease the depth d at which Algorithm 5.1 discovers a solution, making it practical for large, complex graphs. We illustrate our ideas in Figure 5.9, which shows how query RQ1 from Figure 5.1(b) is discovered (only the computation of its lattice root is shown): Figure 5.9(a) illustrates the execution of Algorithm 5.1, which requires $d = 5$.[2] The *generalized stars* optimization in Figure 5.9(b) reduces this to $d = 3$. And the *intersection* optimization in Figure 5.9(c) requires only $d = 2$.

Generalized Stars: Recall that a star was formed by merging a set of k starred nodes, one each from a tree in $Trees(Out.A_i), 1 \leq i \leq k$. A generalized star is formed by merging $m \geq k$ starred nodes from k trees, one from each $Trees(Out.A_i)$, such that each tree contains at least one of the m starred nodes. For example, in Figure 5.9(b), we merge $PS2$, $PS3$ and $PS4$, where $PS2$ and $PS3$ are from the same tree. The result is a star with parallel paths between the star center and some of the tree roots.

We implement this by changing the star computation in Step 3 as follows: For each list Ł in $Valid(S)$, enumerate all different subsets $\mathcal{S} = \{S_1, \ldots, S_m\} \subseteq$ Ł, $m \geq k$, where the nodes in $\mathcal{S}$ are from exactly k trees, and each tree is in a different set $Trees(Out.A_i), 1 \leq i \leq k$. Form a star as a union of paths between

[2]The graph in Figure 5.1(b) is obtained from the star in Figure 5.9(a) via $Merge(S2, S3)$, $Merge(S1, S4)$, $Merge(PS2, PS3)$, $Merge(PS1, PS4)$, and $Merge(P1, P2)$.

each $S_i \in \mathcal{S}$ and its tree root; merge the nodes $S_1, \ldots, S_m$ into a single node S. In particular, this procedure also generates all stars as before (for $m = k$).

Intersection: Note that we generate portions of the graph in Figure 5.1(b) during early iterations: We generate the upper path *S1-N-S2* at $d = 1$ as a star query Q_1 of center N; see Figure 5.9(c). Similarly, the path *S1-PS1-P-PS2-S2* is a star query Q_2 of center P, generated at $d = 2$. Neither Q_1 nor Q_2 pass testing, since each generates a superset of Out. However, $Q_1 \cap Q_2 = Out$. Here, $Q_1 \cap Q_2$ denotes the intersection of the outputs of Q_1 and Q_2. At a graph level, $Q_1 \cap Q_2$ corresponds to "gluing" the graphs of Q_1 and Q_2 by merging their respective projection tables (in this example, $S1$ and $S2$), while keeping all other nodes distinct.

This suggests the following optimization: At the end of each iteration d, if a solution is discovered, return it. Otherwise, compute the intersection of all inclusion-minimal query outputs generated during Step 4 of any prior iteration (including d), which are a superset of Out (the inclusion-minimal condition implies that if $Q1$ is an ancestor of $Q2$ in a lattice, and they both compute a superset of Out, then $Q1$ is not used in the intersection). We use Bloom filters for large query outputs to speed up computation. If the intersection is equal to Out, then we have a solution. We reduce the complexity of this solution via the following greedy approach: Eliminate one query at a time from the intersection, as long as the resulting intersection is equal to Out. Queries are eliminated in decreasing order of their complexity (i.e., their number of nodes). Although the greedy approach does not necessarily generate the minimum complexity query, it is likely to work well in practice. This is because the problem of computing the smallest complexity queries whose intersection is equal to Out is equivalent to Set Cover (we omit the proof for lack of space). It is well known that Set Cover is NP-Hard, but the greedy algorithm achieves a good approximation.

Remark: In Figure 5.9 the intersection optimization is sufficient to discover the query graph from Figure 5.1(b). However, for more complex queries we need both generalized stars and intersection to reduce d. For example, the (undirected) graph Q in Figure 5.8(a) can be generated at depth $d = 3$ as the intersection of two graphs: Graph $Q1$ is the subgraph over vertices $\{u, v, d, e\}$, and is computed at $d = 2$ in the lattice of $Star(e)$. Graph $Q2$ is the subgraph over vertices $\{u, v, a, b, c, g, f\}$ and is computed at $d = 3$ in the lattice of the generalized star $GenStar(f)$ in which f has two parallel paths to u.

5.4.2 Bounding TID list sizes

We now propose a second type of optimizations that bounds the size of the TID lists to reduce the execution time of Step 2.

Wild card: Step 2 instantiates the lists $TID(S\text{i}, \theta)$ by executing joins along tree edges. When a join is in the pk/fk direction, the size of the child node's TID list generally increases. The increase can be significant if the parent TID contains tuples with high fanout, and it grows geometrically as we traverse down the tree.

We propose the following optimization: Let α be a threshold value. While executing Step 2, whenever $|TID(S\text{i}, \theta)| \geq \alpha$ for some node Si, we set $TID(R\text{j}, \theta) = \{wc\}$ for all nodes Rj in the subtree rooted at Si (including Si). Here, wc denotes a special tid (called *wild card*) that matches any other tid by definition. Thus, if $TID(S\text{i}, \theta) = \{wc\}$ and $TID(S\text{j}, \theta) = \{tid_1, tid_2, \ldots\}$, then Si and Sj are mergeable. To enforce this we add Si to $IL_a(tid_r, \theta)$ (and $IL(tid_r, \theta)$ if defined) for all tid_r stored in S nodes. We define $IL(wc, \theta) = IL_a(wc, \theta) = \emptyset$.

Wild cards have two advantages: First, we do not need to execute any instance-level joins below a wild card. Instead, we just traverse the in-memory subtree structure and set the TID's appropriately. Second, computing the lists in $Valid(S)$ and $Merge(S)$ is much faster: (a) S nodes that contain wild cards occur by default in each list of $Valid(S)$, resp. $Merge(S)$; and (b) nodes without wild cards have TID's of size at most α. This implies an upper bound of $O(n\alpha^2)$ for computing each $Valid(S)$ (where n is the number of nodes in all trees). We study the impact of different α values in Section 5.5.

Blacklisted values: One drawback of wild cards is that we might generate more candidate stars and larger lattices: the wild cards may lead us to declare more and larger sets of nodes as mergeable. To mitigate this drawback, we would like to select output tuples from Out such that at least one tuple acts as negative witness for any pair of nodes that are not mergeable. Clearly, if two selected tuples θ and θ' have $\theta.A_i = \theta'.A_i = v$, then θ and θ' instantiate the same TID's in all trees of $Trees(Out.A_i)$. In particular, if θ creates wild cards, so does θ', and neither can be a negative witness for the wild card nodes. To avoid this, whenever a selected tuple θ creates at least one wild card in a tree of $Trees(Out.A_i)$, we blacklist $v = \theta.A_i$ for column $Out.A_i$. We maintain blacklisted values for each output column, and adapt the random sampling from Out so that we only select tuples θ' whose values are not currently blacklisted for their respective columns (if v is blacklisted for $Out.A_i$ but not for $Out.A_j$, we may still select tuples θ' with $\theta'.A_j = v$).

Value blacklisting is useful even in the absence of wild cards. By blacklisting values that generated large TID lists, we can speed up the computation of Step 2 for subsequent tuples θ'.

5.5 Experimental Evaluation

We evaluate our approach on the TPC-H benchmark database, whose schema is shown in Figure 5.1(a). We generate an instance using a tool [4] which creates skewed column distributions (unlike the standard generator, which creates a uniform distribution). This is because we wish to study the effect of different tuple fanouts and join selectivities on the performance of our algorithm. The generated TPC-H instance has size 140MB.

Experimental Setup We conducted two sets of experiments. The first set is over the TPC-H querylog - as we show in Section 5.5.1, we reverse-engineer all but one query within a small running time. The second set is created by us in order to study the effects of various parameters and optimizations, and is described in Section 5.5.2. Our algorithm is implemented in C++ and the experiments were performed on a Windows server with a Quad-Core AMD Opteron 2.3 GHz CPU and 128GB RAM running MySQL.

Methodology We execute each query Q over our TPC-H instance to produce a table Out, then call Algorithm 5.1 with inputs Out and the TPC-H database. We set $d_{max} = \infty$, i.e., we increment d until we discover a query Q' such that $out(Q') = Out$. (The simpler variant that outputs any query with $out(Q') \supseteq Out$ is also discussed in relevant cases.) We refer to this process as running the algorithm for query Q; however, we emphasize that our algorithm is not given any information on the original query Q, other than its output. The parameter settings we used are summarized in Table 5.1; the default values are shown in bold.

5.5.1 TPC-H Queries

There are 22 queries provided with the TPC-H benchmark – which we denote TQ1,. . .,TQ22. Most of them contain aggregates, arithmetic expressions,

Parameter	Setting
Schema-level pruning	no, **yes**
Lattice testing	top-down, **bottom-up**
Nr. of random tuples in Θ	1, 2, 3, 4, **5**
Wild card threshold α	100, 1K, 10K, **100K**, 1M
Value blacklisting	no, **yes**

Table 5.1: Experiment parameters and settings

groupby statements, selection conditions etc. - which go beyond the scope of this paper. Therefore, we modify the TPC-H queries by dropping all such operators. However, we maintain all the join conditions, i.e., the TPC-H query graphs are unchanged. The projection tables are all those tables whose columns appeared in the original query as projection columns, or in any selection conditions or groupby statements. Note that the more projection tables we use, the more columns in table *Out* and the more trees and stars we have to examine.

# Joins	TQ	Runtime (s) min-max	d_{max} min-max	# Graphs min-max	Discovered?
1	4,12-15, 17,19,22	0.8 – 34	1	1	Yes
2	3, 11, 16, 18	0.9 – 11.2	1	1	Yes
3	10	6.4	2	1	Yes
4	2, 20	8.4 – 39.7	1 – 2	1	Yes
5	7, 9, 21	11.9 – 135.8	2 – 3	1 – 2	Yes
6	5	–	–	–	No
7	8	202.2	3	3	Yes

Table 5.2: Results on TPC-H queries (grouped by number of joins)

Of the 22 queries, TQ1 and TQ6 are selection queries from a single table, i.e., with no joins. We discover them at depth $d = 0$ within 1 second. Our results on the remaining queries are shown in Table 5.2. For each query, we measure the running time, number of tested candidate graphs, and the depth d_{max} at which a solution is discovered. We group the queries by their number of joins, and report the min/max values of these measures within a group. The second column in Table 5.2 shows the query id's in that group.

The last column indicates whether our algorithm discovered the original query: all queries in all groups are efficiently re-discovered by our algorithm, with the exception of TQ5 (group 6). The output of TQ5 only contains 4 tuples. Thus, any query graph that generates a superset of these 4 tuples is a candidate graph: there are 21 candidates at depth 2 and 70 at depth 3. Our algorithm can efficiently discover them, but testing takes too long (we stopped it after several hours). We manually checked that TQ5 was among the candidate graphs generated at depth 3. Thus, given sufficient time, the algorithm would have discovered it. For all other TPC-H queries, our method was successful within less than 3.5 minutes.

5.5.2 Our Queries

We study different aspects of our algorithm via the queries in Table 5.3; the projection tables are underlined in the graphs, and their projection attributes are shown in column "Projection." The queries range from simpler ones (Q1 and Q2) to more complex (Q3 through Q6). We deliberately included 4 queries that all have a pair of *Supplier* projection tables, in order to illustrate the challenges of distinguishing among queries whose outputs are similar, but not identical.

We study the benefit of schema-level pruning via query Q1. For more complex queries, this benefit decreases, since at higher depths d most tables are star centers. We use Q4 to investigate the effects of instance-level pruning. The reason is that $Q4$ contains two joins $S - L$ and the tuple fanouts along this join can be very large, leading to large TID lists. Thus, we also study the effects of wild card and value blacklisting via $Q4$. Queries Q2 and Q3 are case studies for lattice exploration and testing, and illustrate two different scenarios. Queries $Q5$ and $Q6$ are then used to study the effects of the intersection optimization.

5.5.3 Schema-Level Pruning

We study the benefits of schema-level pruning in terms of the number of nodes/edges that are deleted from the trees, as well as the savings in running time. We report results for query $Q1$. Figure 5.10(a) shows the number of nodes and edges in schema-level trees, before and after schema-level pruning: we prune 19 out of 33 tree nodes and 12 out of 19 edges. As a sanity check, Figure 5.10(a) also includes the number of lattices and graphs before and after schema-level pruning. Note that the number of candidate graphs is the same in

QID	Query Graph	Projection	Description	Discovered?
Q1	$\underline{C} \leftarrow \underline{O} \leftarrow \underline{L}$	C.c_custkey O.o_orderdate L.l_shipdate	Find the customers and the order/ship dates of their orders.	No. Simpler query discovered instead.
Q2	$\underline{S1} \rightarrow N \leftarrow \underline{S2}$	S1.s_suppkey S1.s_name S2.s_suppkey S2.s_name	Find all pairs of suppliers located in the same nation.	YES
Q3	$$\begin{tikzcd} & & L \arrow[d] & & \\ \underline{P1} & PS1 \arrow[l] \arrow[r] & S & PS2 \arrow[l] \arrow[r] & \underline{P2} \end{tikzcd}$$	P1.p_name P2.p_name	Find all pairs of parts that are supplied by the same supplier who has supplied line items.	YES
Q4	$\underline{S1} \leftarrow L1 \rightarrow O \leftarrow L2 \rightarrow \underline{S2}$	S1.s_name S2.s_name	Find all pairs of suppliers supplying in the same order.	YES
Q5	$$\begin{tikzcd} & & N & & \\ \underline{S1} \arrow[urr] & PS1 \arrow[l] \arrow[r] & P & PS2 \arrow[l] \arrow[r] & \underline{S2} \arrow[ull] \end{tikzcd}$$	S1.s_name S2.s_name	Find all pairs of suppliers located in the same nation and supplying the same part.	YES
Q6	$$\begin{tikzcd} & N1 \arrow[r] & R & N2 \arrow[l] & \\ \underline{S1} \arrow[ur] & PS1 \arrow[l] \arrow[r] & P & PS2 \arrow[l] \arrow[r] & \underline{S2} \arrow[ul] \end{tikzcd}$$	S1.s_name S2.s_name	Find all pairs of suppliers located in the same region and supplying the same part.	YES

Table 5.3: TPC-H query set. (The projection tables are underlined.)

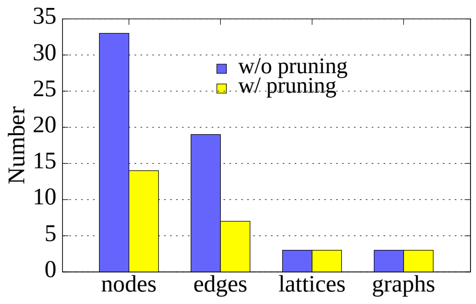
	Number	avg Fanout	max Fanout
Θ_1	5	72.0	280
Θ_2	5	2151.1	34720

Table 5.4: Characteristics of Θ_1 and Θ_2.

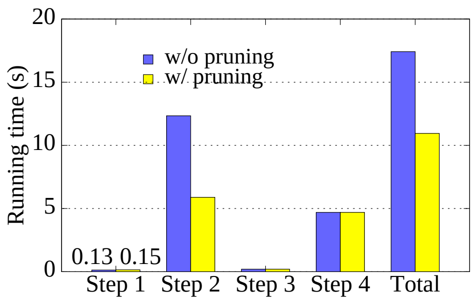
either case, i.e., pruning only deletes unnecessary tree nodes and edges, without affecting correctness.

Figures 5.10(b) and 5.10(c) show the reduction in the running time of each step, starting from two different sets of random tuples, Θ_1 and Θ_2. Exploring each set generates widely different sizes of TID lists, as indicated in Table 5.4. The table shows the average/max fanout over all tuples explored in all trees during Step 2.

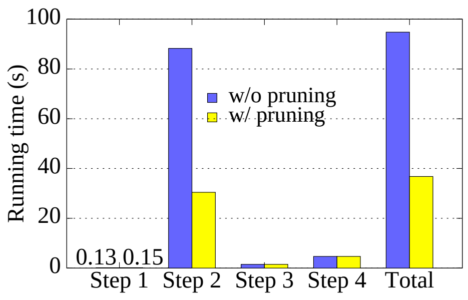
As expected, the total time is dominated by the database-related operations, i.e., the instance-level exploration (Step 2) and the testing phase (Step 4). The nodes/edges that are not present in the final stars would not have passed instance-level pruning anyway. Thus, removing them at schema-level only saves time in Step 2, but does not affect Steps 3 and 4. However, the savings for Step 2 are significant: its running time decreases by 52%, resp. by 65%, on tuple sets Θ_1, resp. Θ_2. As a result, we save 37%, resp. 61%, of



(a) Data structures.



(b) Running times: tuple set Θ_1.



(c) Running times: tuple set Θ_2.

Figure 5.10: Effects of schema-level pruning for Q1.

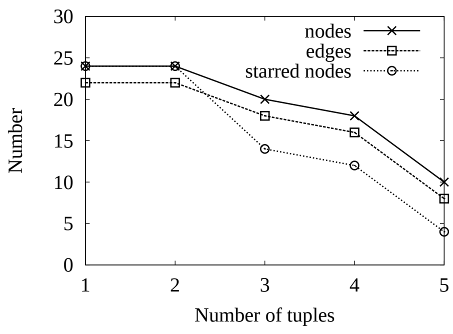
total processing time on each set. Clearly, the higher the fanout of a set Θ_i, the more we benefit from pruning. The overhead of schema-level pruning is negligible (0.02 seconds). We conclude that schema-level pruning is very fast and should always be applied. Its benefits decrease for more complex queries, but are very significant for simpler ones.

Interestingly, our algorithm was able to find a simpler SQL query (1 join, 2 tables) that generates the same output as $Q1$ (2 joins, 3 tables). The computed query, shown below, is clearly correct, since `O.o_custkey` is a foreign key of the originally projected primary key `C.c_custkey`:

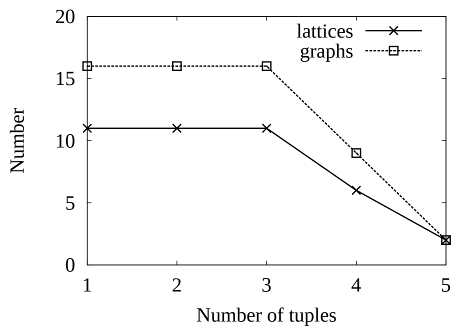
```
SELECT N0.o_custkey, N0.o_orderdate, N1.l_shipdate
FROM ORDERS N0, LINEITEM N1
WHERE N0.o_orderkey = N1.l_orderkey
```

5.5.4 Instance-level Pruning

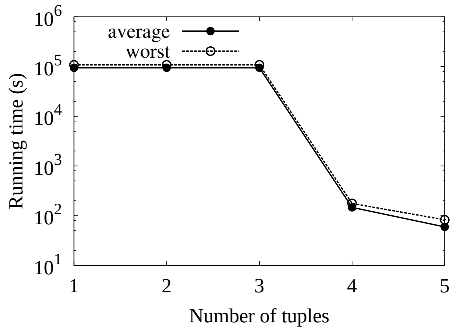
Recall that instance-level pruning occurs when output tuples in Θ are negative witnesses for tree edges and nodes, as well as for starred sets (Section 5.3.2). We now study the effect of varying the size of Θ and report our results for query



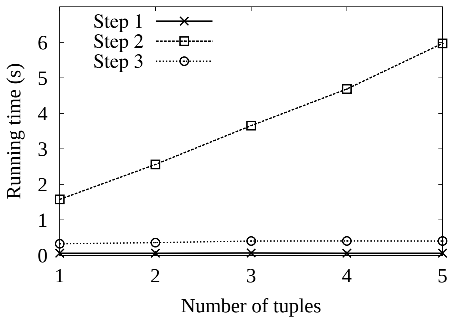
(a) Tree statistics.



(b) Lattices and candidate graphs.



(c) Running time of Step 4.



(d) Running time of Steps 1-3.

Figure 5.11: Instance-level pruning for Q4, as a function of $|\Theta|$.

Q4. Over 10 random draws of Θ, we report the worst case: this occurred when the first two tuples in Θ did not yield any instance-level pruning. However, tuples 3, 4 and 5 each reduced the number of nodes, edges and starred nodes; refer to Figure 5.11(a). For $|\Theta| = 5$, we are able to prune 14 out of 24 nodes, 14 out of 22 edges and reduce the number of starred nodes from 24 to 4. This is a significant reduction, especially for starred nodes, which results in fewer lattices and candidate graphs. Figure 5.11(b) shows that tuples 4 and 5 reduce the number of lattices: for $|\Theta| = 5$, the number of lattices decreases from 11 to 2, and the number of candidate graphs from 16 to 2. We note that further reductions are not possible, because each of the remaining two candidate graphs generates a superset of *Out*. Therefore, any additional tuple in Θ will validate both queries. This is the reason we only report results up to $|\Theta| = 5$.

For Q4, the running time of the algorithm is highly dominated by the testing in Step 4. Thus, reducing the number of candidate graphs has significant impact. Figure 5.11(c) shows that, by using 5 random tuples in Θ, we are able to reduce the running time by 3 orders of magnitude! This is because some of the graphs invalidated by the additional tuples are among the most expensive to test, since they behave like cross-products. We report both worst and average cases (over all possible testing orders). Recall that we stop once we discover the query that generates table *Out*.

The cost of exploring additional tuples can be high, but we use our wild card optimization to bring it back down. In this experiment, we used the default wild card threshold $\alpha = 100K$. Experiments with other thresholds are discussed in the next subsection. Figure 5.11(d) shows that for $\alpha = 100K$, the running time of Step 2 increases linearly with the number of tuples in Θ (as expected), but the cost is only about 1.2 seconds per additional tuple. This incremental cost was relatively consistent across all the queries we tried, and depends mostly on the characteristics of the database instance (e.g., average tuple fanout). For example, Figures 5.10(b) and 5.10(c) imply a cost of about 1 second, resp. 5 seconds, per tuple for Q1 (after schema-level pruning; recall that $|\Theta_1| = |\Theta_2| = 5$). For the sake of completeness, Figure 5.11(d) also shows the running times of Steps 1 and 3, which are negligible.

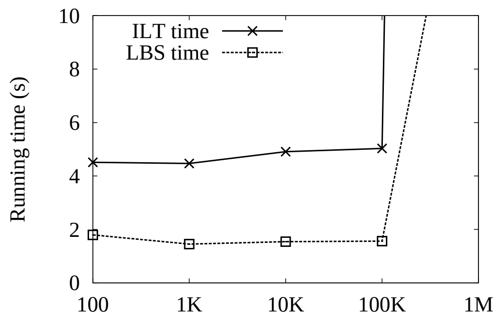
Finally, we mention that we observed overall reductions in running time for most of the queries we tried, when selecting between 3 and 5 random tuples in Θ. The savings varied from query to query. But, given the small overhead in Step 2, versus the much larger potential benefit in Step 4, we conclude that we should always select multiple tuples in Θ. The size of Θ can be adjusted dynamically, using heuristic estimates for the cost/benefit of each additional tuple.

5.5.5 Optimizations: bounding TID size

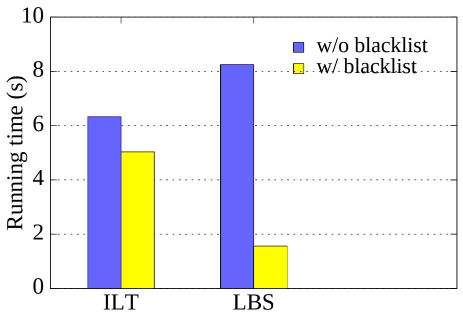
Instance-level pruning is closely correlated to the optimization techniques described in Section 5.4.2 that bound the size of the TID lists. Hence, we report experiments on these optimizations on the same query Q4 as above. The two sets of experiments should be evaluated together.

Wild card

We study the effect of varying the wild card threshold α on the running time of Step 2. The result (using 5 tuples in Θ) is plotted in Figure 5.12(a). We report two statistics: ILT is the time for instance-level tree traversal, i.e., retrieving the



(a) Wild card threshold.



(b) Value blacklisting.

Figure 5.12: Bounding TID sizes.

tuples in all TID lists from the database; and LBS is the time for computing the lattice-building data structures (i.e., the $Valid$ and $Merge$ lists). For $\alpha = 100$, the LBS time is slightly higher than for α=1K–100K, since we increase the number and size of mergeable lists. The cost of ILT increases very slowly from $\alpha = 100$ to $\alpha = 100K$. At the other extreme, $\alpha = 1,000,000$ causes a huge increase for both ILT and LBS. For this threshold, ILT becomes the most expensive part of the algorithm (10 minutes), dominating even testing (113 seconds).

However, a threshold of up to 100K results in a combined ILT and LBS time of about 6 seconds, or an average of 1.2 seconds per tuple in Θ. Thus, we can maintain a small incremental cost per additional tuple in Θ by using the wild card optimization and appropriate thresholds.

Value blacklisting

We blacklisted the values that incurred large fanouts during the instance-level traversal, to ensure that we do not increase the number of lattices and candidate graphs. Thus, the testing time remained the same as in Figure 5.11(c) (for $|\Theta| = 5$). We also applied blacklisting in the absence of wild cards, to observe its effects on the ILT and LBS running times; see Figure 5.12(b). In this experiment, we randomly chose 10 different sets Θ. Out of these, 4 contained tuples that created blacklisted values; there were 2 blacklisted values. We report the average over these 4 sets of results. The running time decreased by 1 second for ILT and 6.5 seconds for LBS.

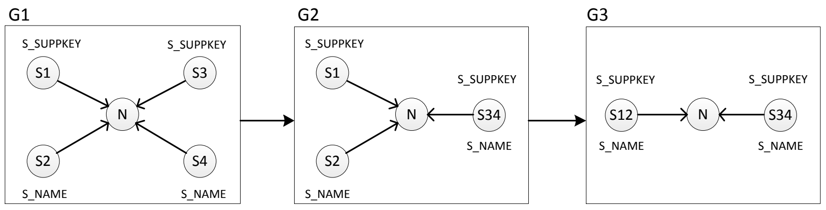


Figure 5.13: Lattice for Q2; $d = 1$.

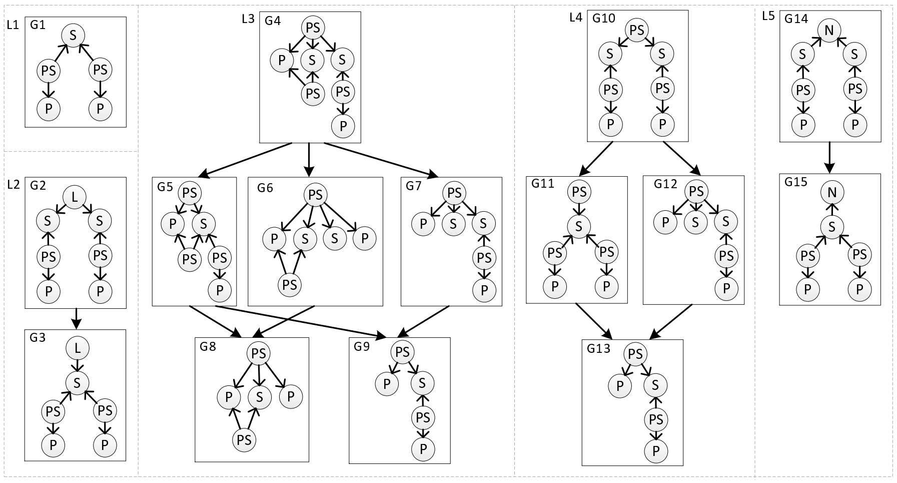


Figure 5.14: Lattices for Q3; $d = 3$.

5.5.6 Lattice Exploration and Testing

Lattices are the key structure for exploring candidate graphs. They not only allow us to discover arbitrary graphs, but also guide the testing step. In this section we illustrate two different scenarios encountered by our algorithm for queries Q2 and Q3, and discuss two testing strategies for each: bottom-up and top-down (Section 5.3.4). Figures 5.13 and 5.14 show the lattices for Q2, resp. Q3, for the depth d where each is discovered. Q2 is discovered at $d = 1$ and has a single lattice with 3 graphs; Q3 is discovered at $d = 3$ and has 5 lattices with 15 graphs. The smaller lattice complexity for Q2 is because it is discovered at smaller depth d, and it has a larger number of output columns. Both factors imply fewer star centers (and lattices).

Q2 asks for all pairs of suppliers located in the same nation and selects both the id and the name of each supplier. Since the output table has four columns, this results in stars with four branches. The algorithm detects one star center at $d = 1$, i.e., table *Nation*. The star is graph G1 in Figure 5.13. It has two potentially valid merges, $Merge(S1, S2)$ and $Merge(S3, S4)$. However, the

Query graph	G1	G2	G3
time	1 h	38.95 s	0.22 s

Table 5.5: Testing time of query graphs for Q2.

resulting graphs are isomorphic, so one of them is eliminated. The grandchild graph is the result of executing both merges, and is the same as Q2. Table 5.5 shows the testing time of the three candidate graphs from Figure 5.13. The testing time is much higher for G1 than for G2 and G3. This is because G1 is a 4-way cross-product query (among subsets of *Supplier*). The more columns a table *Out* has, the more expensive it is to test its stars. However, our lattice exploration discovers simpler descendant graphs when tables are truly mergeable. Bottom-up testing returns the correct query G3 without exploring the others, in a total time of 9 seconds; most of this is spent in Step 2. By contrast, top-down testing requires more than an hour, and tests all three queries.

Q3 generates all pairs of parts that are supplied by the same supplier, who also supplies at least one line item (not every supplier instantiates the join to table *LineItem*). Figure 5.14 shows the 5 lattices generated by our algorithm (labeled by Li), with a total of 15 graphs (Gi). (For better readability, we removed the numbers from node labels). Note that G1, G9 and G13 are isomorphic, so only one of them is tested. There are two graphs, G2 and G3, that return exactly table *Out*. However, G3 has lower complexity than G2. Thus, while either graph is correct, G3 is the better answer (it is also the same as Q3).

Table 5.6 shows the candidate queries that are tested by the two strategies. Graph G1 is considered by both strategies. However, it is also generated during the prior iteration $d = 2$. Hence, we include it in Table 5.6 but not in Table 5.7, which shows worst and average case statistics for both strategies. Table 5.7 is computed as follows. For either strategy, queries that do not have an ancestor/descendant relationship in some lattice are tested in random order. We ignore G1 (tested at $d = 2$), and G9 and G13 (isomorphic to G1). The average case statistics are computed over all permutations of the remaining graphs in each strategy, with the top-down strategy stopping after G2 is tested, and the bottom-up strategy stopping after G3. Thus, the bottom-up strategy tests 3 graphs in the worst case, and 2.2 on the average. By contrast, the top-down strategy tests 4 (3.0) graphs in the worst (average) case. Moreover, the running time of the top-down approach is much longer than for bottom-up in each case. Since top-down also returns the more complex answer G2, we conclude that the bottom-up strategy is more efficient and likely to find simpler answers.

top-down	G1	G2	G4	G14	G10	
bottom-up	<u>G1</u>	G3	G8	G15	<u>G13</u>	<u>G9</u>

Table 5.6: Tested candidate graphs for Q3 (note: G1=G9=G13).

	Worst Case		Average Case	
	number	time(s)	number	time(s)
top-down	4	18918.0	3.0	13051.1
bottom-up	3	480.5	2.2	331.5

Table 5.7: Number of graphs and testing time for Q3, at $d = 3$.

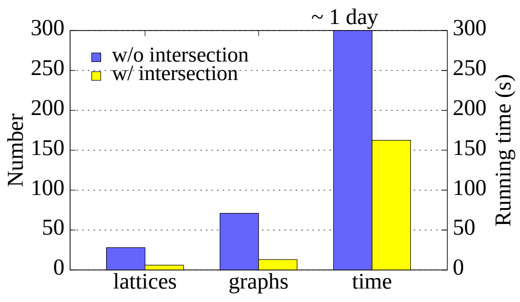
Problem Variants For the superset semantics, a graph passes testing if it generates a superset of *Out*. In this case, graph G1 passes the test at $d = 2$ and the algorithm returns.

In another variant, we want all graphs that pass the test (either for exact match or superset semantics) at $d = 3$. In this case, the bottom-up strategy tests only one more graph than above, i.e., graph G6. This is because G8 does not generate a superset of *Out*, so its ancestor G6 is a potential candidate. Graphs G9 and G13=G15=G1 do generate strict supersets of *Out*, so we need not test their ancestors: they pass the test for the superset semantics, and fail it for the exact match. The top-down strategy, by contrast, needs to test all graphs for this variant.

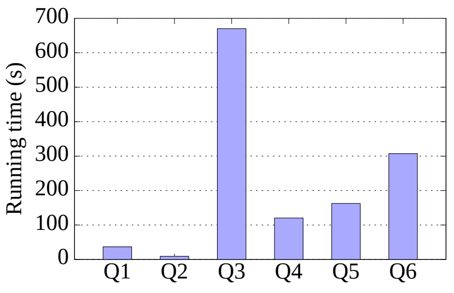
5.5.7 Optimizations: decreasing depth d

We investigate the effects of the intersection optimization for queries Q5 and Q6. A simple analysis shows that this optimization reduces d from 5 to 2 for Q5 (see also Figure 5.9), and from 6 to 2 for Q6. Figure 5.15(a) shows more details for Q5: with intersection, the number of lattices is reduced from 28 to 6, and the number of candidate query graphs decreases from 71 to 13.

For both queries, the algorithm without intersection could not finish after one day. However, with intersection, the running times were reduced to 162.5 seconds for Q5, and 305 seconds for Q6. Figure 5.15(b) shows the running times



(a) Details for Q5.



(b) Overall running times.

Figure 5.15: Effects of intersection for Q5 and Q6.

for all the queries. (Note: for Q3, the time is higher than in Table 5.7 since it includes the time for iterations $0 \leq d \leq 2$). Since Q1 and Q2 are discovered at $d = 1$, Q4–Q6 are discovered at $d = 2$, and Q3 is discovered at $d = 3$, Figure 5.15(b) also shows a trend where the running time increases by a factor of 2 to 5 when incrementing d by 1. This is further evidence that reducing d is essential for making our approach able to compute complex graphs.

5.6 Summary

In this chapter, we proposed a new approach for reverse engineering arbitrary join queries. Our approach relies on a novel characterization of graphs, based on the notions of stars and merge sets, which may be of independent interest. In our experiments over TPC-H we were able to compute complex queries due to a variety of proposed optimizations which make our method scale to complex graphs. Our algorithm is quite general, and can be used for several problem variants and application scenarios.

This work has been published as a full research paper in *2011 ACM SIGMOD/PODS Conference* [58].

CHAPTER 6

Conclusion and Future Work

6.1 Conclusion

Complex databases with poor or missing documentation result in great difficulties for users to understand and extract useful information from the data. In this thesis, we designed automatic and purely data oriented approaches to discover helpful information that aids the users in understanding the relationships between relational tables and columns.

Our first goal was to design an effective approach to discover foreign key constraints in relational databases. In Chapter 3, we have introduced the notion of *Randomness* and showed that it can be used effectively to reduce the false positives produced by inclusion test. We also provided an efficient approximation algorithm which uses quantile summaries for evaluating randomness over a large set of columns. In addition, we designed an I/O efficient algorithm which requires only two passes over the data for outputting the final list of foreign/primary key pairs. This leads to a novel and effective foreign key discovery rule that is applicable to relational databases in practice. Notably, multi-column foreign keys are also addressed in our work, which have not been considered by previous work.

The next objective of this thesis was to provide a solution to discovering semantically equivalent attributes. Towards this goal, we have proposed in Chapter 4 a robust, unsupervised solution that was able to efficiently and accurately identify the semantic correspondence between relational columns. Our solution is purely data oriented. We do not rely on any form of external knowledge about the data. Through an extensive experimental study over real and benchmark datasets, we have shown our approach was able to correctly identify attributes with very high accuracy. Having the efficiency and accuracy, our approach can be an invaluable tool for data integration and schema matching applications, besides for assisting users in understanding the relational data.

We also aimed to design a principled solution to mine the relationship between a given SQL answer table to the remaining tables in the database. We focused on join queries. We have proposed in Chapter 5 a novel and quite general approach that can be used for several problem variants and application scenarios. We have introduced the notions of stars and merge sets and proved that any graph can be characterized by the combination of the stars and a series of merge steps over the stars. In contrast with prior work where specific restrictions are imposed on the structure of the query graph, we have shown that our approach could discover queries with arbitrary graphs. We have also designed a variety of optimizations that significantly improve the efficiency, making our approach scale to very complex graphs and applicable in practice.

6.2 Future Work

In the future, we plan to investigate whether the techniques proposed in Chapter 3 and 4 can be extended to discover multi-column attributes (for example when customer names are expressed as separate first/last name columns).

Further, we have identified the use of randomness and EMD to create distribution clusters has one limitation, i.e. for the scenarios where horizontally partitioned attributes (e.g., telephone numbers based on locations) appear. In the future, we would like to explore whether information theoretic techniques can be used to solve this problem.

Another possible direction is to integrate our approach presented in Chapter 5 with the method of [56], which reverse engineers the selection conditions of a query. Together, it would enable the discovery of general SPJ queries. We also note that reverse engineering a query that contains arbitrary arithmetic expressions is PSPACE-hard, so SPJ queries are the best we can hope to achieve with a general methodology.

In addition, we would also like to extend our work to handle OLAP queries which contain group-by and aggregations. Reverse engineering the aggregation queries with selection conditions is not trivial even if there is no arbitrary arithmetic expression inside the aggregation. Given that there may exist selection conditions, we can have multiple guesses for one aggregation value. For example, a SUM over a small subset of tuples is also likely to be AVE, MIN or MAX over another subset of tuples. Designing an efficient algorithm to explore all the possibilities become challenging.

Bibliography

[1] CPLEX optimizer. http://www.ibm.com/software/integration/optimization/cplex-optimizer.

[2] DBLP database. http://dblp.uni-trier.de/xml.

[3] IMDB database. http://www.imdb.com/interfaces.

[4] A tool for generating skewed data distributions for TPC-H data from Microsoft Research. ftp://ftp.research.microsoft.com/users/viveknar/TPCDSkew.

[5] TPC-H benchmark. http://www.tpc.org/tpch.

[6] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[7] B. Ahmadi, M. Hadjieleftheriou, T. Seidl, D. Srivastava, and S. Venkatasubramanian. Type-based categorization of relational attributes. In *EDBT*, pages 84–95, 2009.

[8] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *Journal of the ACM*, 55(5), 2008.

[9] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. *Machine Learning*, 56:89–113, June 2004.

[10] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz. Efficiently detecting inclusion dependencies. In *ICDE*, pages 1448–1450, 2007.

[11] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*, pages 199–210, 2007.

[12] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[14] L. Blunschi, C. Jossen, D. Kossmann, M. Mori, and K. Stockinger. Soda: Generating sql for business users. *PVLDB*, 5(10):932–943, 2012.

[15] A. Broder. On the resemblance and containment of documents. In *SEQUENCES*, pages 21–30, 1997.

[16] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[17] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *PODS*, pages 171–176, 1982.

[18] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, page 5, 2006.

[19] E. F. Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971.

[20] Edgar F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[21] E. Cohen and H. Kaplan. Leveraging discarded samples for tighter estimation of multiple-set aggregates. *Joint Intl. Conf. on Measurement and Modeling of Comp. Syst.*, pages 251–262, 2009.

[22] J. Considine, M. Hadjieleftheriou, F. Li, J. W. Byers, and G. Kollios. Robust approximate aggregation in sensor data management systems. *TODS*, 34(1), 2009.

[23] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[24] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley & Sons, Inc, 2003.

[25] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *SIGMOD*, pages 240–251, 2002.

[26] R. Dhamankar, Y. Lee, A. Doan, A. Y. Halevy, and P. Domingos. iMAP: Discovering complex mappings between database schemas. In *SIGMOD*, pages 383–394, 2004.

[27] H. H. Do and E. Rahm. COMA - a system for flexible combination of schema matching approaches. In *VLDB*, pages 610–621, 2002.

[28] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529, 1999.

[29] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[30] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *SIGMOD*, pages 58–66, 2001.

[31] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: Ranked keyword searches on graphs. In *SIGMOD*, pages 305–316, 2007.

[32] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[33] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE. http://www.cs.helsinki.fi/research/fdk/datamining/tane/.

[34] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE*, pages 392–401, 1998.

[35] J. Kang and J. F. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, pages 205–216, 2003.

[36] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *VLDB*, pages 180–191, 2004.

[37] A. Koeller and E. A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *ICDE*, pages 683–685, 2003.

[38] S. Lopes, J.-M. Petit, and Lotfi L. Efficient discovery of functional dependencies and armstrong relations. EDBT, pages 350–364, 2000.

[39] S. Lopes, J.-M. Petit, and F. Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems*, 27(1):1–19, 2002.

[40] J. Madhavan, P. A. Bernstein, and E. Rahm. Generic schema matching with cupid. In *VLDB*, pages 49–58, 2001.

[41] F. De Marchi, S. Lopes, and J.-M. Petit. Unary and n-ary inclusion dependency discovery in relational databases. *Journal of Intelligent Information Systems*, 32(1):53–73, 2009.

[42] F. De Marchi and J.-M. Petit. Zigzag: a new algorithm for mining large inclusion dependencies in databases. In *ICDM*, pages 27–34, 2003.

[43] S. Melnik, E. Rahm, and P. A. Bernstein. Rondo: A programming platform for generic model management. In *SIGMOD*, pages 193–204, 2003.

[44] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB*, pages 122–133, 1998.

[45] S. Peleg, M. Werman, and H. Rom. A unified approach to the change of resolution: space and gray-level. *TPAMI*, 11(7):739–742, 1989.

[46] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, pages 73–84, 2012.

[47] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: The power of rdbms. In *SIGMOD*, pages 681–694, 2009.

[48] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying communities in relational databases. In *ICDE*, pages 724–735, 2009.

[49] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[50] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill international editions: Computer science series. McGraw-Hill Education, 2002.

[51] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.

[52] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.

[53] S. Siegel and N.J. Castellan. *Nonparametric statistics for the behavioral sciences*. McGraw–Hill, Inc., second edition, 1988.

[54] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: Efficient and scalable discovery of composite keys. In *VLDB*, pages 691–702, 2006.

[55] Transaction Processing Performance Council (TPC). TPC benchmarks. http://www.tpc.org/.

[56] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.

[57] C. Wyss, C. Giannella, and E. L. Robertson. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances. In *DaWaK*, pages 101–110, 2001.

[58] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, pages 809–820, 2013.

[59] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *PVLDB*, 3(1):805–814, 2010.

[60] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. Automatic discovery of attributes in relational databases. In *SIGMOD*, pages 109–120, 2011.