View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by ScholarBank@NUS

QUERY FROM EXAMPLES

LI HAO

NATIONAL UNIVERSITY OF SINGAPORE

2016

DOCTORAL THESIS

QUERY FROM EXAMPLES

Author:
LI HAO
(B.Eng., Nankai University)

Supervisor:
Associate Professor:
CHAN Chee Yong

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2016

NUS
National University
of Singapore

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Li Hao

June, 2016

ACKNOWLEDGMENT

I would like to express the deepest appreciation to my supervisor, Professor Chan Chee Yong, whose expertise, understanding, and patience added considerably to my graduate experience. Without his guidance and persistent help, I would not have finished my thesis. During my Ph.D. study, I encountered many different problems. Prof. Chan always gave me so much support and strength to conquer them. With vast knowledge and skills, he patiently guided me to build interesting ideas, improve my writing skills and my research capability. In my personal life, he is also very considerate and generous. He cared about my health and helped me so much to solve the financial problems. I am really grateful to have him as my supervisor.

I would also like to thank Professor David Maier, who gave me many useful suggestions and insightful comments for my first piece of work.

Special thanks go out to my wife, Huang Xiaocheng, who is smart, funny, supportive and truly makes a difference in my life. Anytime I have doubts, she can always give me strength and made me a better person. I'm so lucky to have her around during my good and bad days.

I would like to thank my thesis committee, Prof. Tan Kian Lee and Prof. Stephane Bressan for their valuable comments on my thesis as well as recommendation letters for my research assistant position.

I would also like to thank all my friends who have made my Ph.D. life more colorful. They are Li Lu, Wang Guoping, Zheng Yuxin, Zeng Zhong, Tang Ruiming, Liu Qing, Li Yuhong, and many others.

Finally, I would like to thank my parents for their support and trust for every decision I made during my Ph.D. life.

CONTENTS

Declaration i

Acknowledgement iii

Abstract xi

1 Introduction 1

1.1 Example-driven Query Construction 2

1.2 Query-based Approach 5

1.3 Schema-based Approach 7

1.4 Thesis Contributions 10

1.5 Thesis Organization 11

2 Literature Review 13

2.1 Query Construction 14

2.2 Example-Driven Systems 16

2.3 Query Generators 17

2.4 Database Generators 20

2.4.1 Reverse Query Processing 20

2.4.2 Query Equivalence Problem 21

2.4.3 Explaining Missing Answers 23

2.5 Query Refinement Problem 23

3 Query-based Approach 25

3.1 Approach Overview 25

3.2 Cost Model 28

3.2.1 Estimation of Number of Iterations 31

3.3 Query Generator 33

3.4 Database Generator 34

3.4.1 Tuple Classes 35

3.4.2 Overview of Approach 37

3.4.3 Algorithm Skyline-STC-DTC-Pairs 38
3.4.4 Algorithm Pick-STC-DTC-Subset 40
3.5 Discussion 44
3.5.1 Queries with Set-based Semantics 44
3.5.2 Queries with Different Join Schemas 45
3.5.3 Database Constraints 46
3.5.4 Supporting More Expressive Queries 46
3.6 Experimental Evaluation 47
3.6.1 Database and Queries 48
3.6.2 Results for Default Settings 49
3.6.3 Effect of Scale Factor β 51
3.6.4 Effect of Time Threshold δ 52
3.6.5 Efficiency of Algorithm 3.4 53
3.6.6 Effect of Number of Candidate Queries 54
3.6.7 Effect of Initial Database-Result Pair 56
3.6.8 Effect of Size & Entropy of Attributes' Active Domains 57
3.6.9 User Study 59
3.7 Conclusion 65

4 Schema-based Approach **67**

4.1 Introduction 68

4.2 Approach Overview 70

4.2.1 Limitation 75

4.3 Handling The Scenario With Positive Partition 76

4.3.1 Algorithm Query-Schema-Generator 76

4.3.2 Algorithm Database-Generator 79

4.3.3 Result Feedback 84

4.4 Handling the Scenario Without Positive Partition 85

4.4.1 Queries with Bag Semantics 87

4.4.2 Queries with Set Semantics 94

4.4.3 Heuristic Solution 96

4.5 Discussion 98

4.6 Experimental Study 99

4.6.1 Datasets and Queries 100

4.6.2 Performance of Schema-based Approach 101

4.6.3 Comparing Query-based and Schema-based approaches 104

4.6.4 User Study 107

4.7 Conclusion 112

5 Conclusions and Future Work **117**

5.1 Contributions 118

5.2 Future Work 118

Bibliography **121**

ABSTRACT

In today's era of Big Data, there is a lot of interest in do-it-yourself data exploration. For example, cloud-based data sharing and analysis platforms are now available which provide a web-based interface for users to pose queries on their uploaded data. However, expressing information needs using database systems often require writing queries in a formal language which is a challenging task for non-expert database users. This has motivated several recent research efforts to help database users with query construction.

Many of the existing approaches require the users to be familiar with the query language: some approaches provide users with a repository of shared queries to facilitate browsing for similar queries, and other approaches provide a recommender functionality to aid users with query construction by suggesting appropriate query snippets based on their partially constructed queries.

In this thesis, we aim to lower the barrier for today's data consumers to utilize database technology for data analysis by investigating an example-driven approach to help users with query construction. Our proposal does not require users to be familiar with any query language; instead, it only requires that the user is able to determine whether a

given output table is the result of his or her intended query on a given input database. To kick-start the construction of a target query Q, the user first provides an example database-result pair (D, R), where R is the desired output table of Q on the database D. As there will be generally multiple candidate queries that transform D to R, our approach winnows this collection by iteratively presenting the user with new database-result pairs that distinguish these candidates. To minimize the user's effort to determine if a new database-result pair is consistent with his or her desired query, our approach strives to make these distinguishing pairs as close to the original (D, R) pair as possible. In this way, our approach is able to identify the user's target query by seeking the user's feedback on a sequence of slightly modified database-result pairs. Except for the initial database-result pair, which is provided by the user, all the subsequent pairs are automatically generated by the system.

We propose two approaches to solve our example-driven method for query construction. The first approach is a query-based approach that leverages existing research on query reverse engineering to generate a set of candidate queries for iterative pruning with the user's feedback. The second approach is a schema-based approach that first identifies the target query schema via user feedback before pruning the candidate queries for the identified target query schema. Our experimental study demonstrates the feasibility and effectiveness of our example-driven approach for query construction.

LIST OF FIGURES

1.1 Overall Architecture of QFE 3
1.2 Employee database and result pair 5
1.3 Employee database and result pair 6
1.4 Employee database and result pair 7
1.5 Employee database and result pair 9
1.6 Employee database and result pair 10
3.1 Overall Architecture of QFE 26
3.2 Queries for Section 3.6.7 56
3.3 Effect of initial database-result pair 58
3.4 UI screen capture 62

4.1 Queries generated by QBO 69

4.2 Overall Architecture of schema-based QFE 71

4.3 Test queries for experiments 101

4.4 User interface screen capture 107

4.5 Total time to find target query (in secs) 112

LIST OF TABLES

3.1 Notation table of Chapter 3 26

3.2 Per-round statistics for scientific database. 50

3.3 Effect of β for baseball database 52

3.4 Effect of δ for scientific database 53

3.5 Performance of Algorithm 4 for scientific database 54

3.6 Performance of Algorithm 3.4 for varying $|SP|$ 54

3.7 Effect of the number of candidate queries on Q_2 55

3.8 Breakdown of first iteration's runing time (in sec) 55

3.9 Properties of datasets and query results 57

3.10 Number of distinct values for attribute A in datasets 58

3.11 Effect of size & entropy of active attribute domain for query Q_1 60

3.12 Effect of size & entropy of active attribute domain for query Q_2 61

3.13 Per-round statistics for queries 63

3.14 Timing results for user study (in secs) 64

4.1 Notation table of Chapter 4 68

4.2 Employee database and result pair 86

4.3 Performance for each target query 102

4.4 Number of candidate query schemas with different selection attributes size 104

4.5 Results of two approaches 105

4.6 Feedback time for AQ_1 (in secs) 109

4.7 Feedback time for AQ_2 (in secs) 110

4.8 Feedback time for AQ_3 (in secs) 111

4.9 Time results of Q-QFE and S-QFE forAQ_1 (in secs) 113

4.10 Time results of Q-QFE and S-QFE forAQ_2 (in secs) 114

4.11 Time results of Q-QFE and S-QFE forAQ_3 (in secs) 115

CHAPTER 1

INTRODUCTION

Given today's ease of collecting large volumes of data and the need for ad-hoc data querying to find information or explore the data, there is growing adoption of relational database systems, beyond the traditional enterprise context, for managing and querying data. For example, in the scientific community, the Sloan Digital Sky Survey (SDSS) Project [1] provides online querying of a large repository of image-based data using SQL queries, and the recent SQLShare Project [36] provides a web-based interface to facilitate scientists posing SQL queries on their uploaded research data. However, many non-expert database users still primarily rely on scripts or files to handle their data. Even though some users can write simple SQL queries, they are not competent enough to express the complicated query intention. Writing SQL queries for such do-it-yourself data exploration remains a challenging task for non-expert users, and this consideration has motivated several recent research efforts to help users with query construction.

One approach to help users with query construction is to provide a repository for users to share their queries and facilitate browsing for similar queries that can be reused, possibly with minor modifications [35, 42]. For example, SQB maintains a sample of popular user queries to facilitate query reuse [42], and SQLShare facilitates browsing and searching of SQL queries posted by users [35].

Another approach is to provide a query recommendation facility. One way is to recommend entire queries based on a user's and other users' past queries recorded in a query log [14]. If they have similar query records, the system will recommend the other users' queries to the current user. Another kind of recommendation is to recommend query snippets for specified SQL clauses (e.g. tables in from-clause, predicates in where-clause) based on the partial query fragment that the user has typed and any past queries authored by the user [41, 55].

Both query browsing as well as query recommendation approaches require the users to be familiar with SQL as they need to be able to read and write SQL queries. They do not take account of users' query intention either, as users can not express their query intention to these approaches accurately. In addition, these approaches may not be applicable if the data being queried belongs to a private database that is used only by a single user.

1.1 Example-driven Query Construction

In this thesis, we propose a novel example-driven approach, called *Query from Examples (QFE)*, that is targeted at less sophisticated users who might be unfamiliar with SQL. Unlike the previous approaches, QFE is a more "user-friendly" approach that only requires that the user be able to determine whether a new given output table is the result of his or her target query on a given input database.

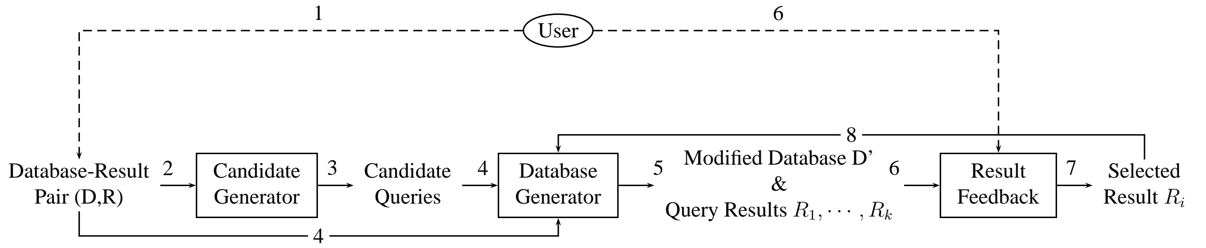


Figure 1.1: Overall Architecture of QFE

To kick-start the construction of a target query Q in QFE, the user first provides an example database-result pair (D, R), where R is the output table of Q when query Q is executed on database D. As there will be many candidate queries that transform D to R, QFE winnows this collection by iteratively presenting the user with new database-result pairs that distinguish these candidates. As for different candidate queries, the database-result pairs could be different. To minimize the user's effort to determine if a new database-result pair is consistent with his or her desired query, QFE strives to make these distinguishing pairs as close to the original (D, R) pair as possible. In this way, QFE is able to identify the user's target query by seeking the user's feedback on a sequence of slightly modified database-result pairs. Except for the initial database-result pair, which is provided by the user, all the subsequent pairs are automatically generated by the system. The overview of QFE architecture is shown in Figure 1.1.

As shown in Figure 1.1, QFE is mainly composed of three components. All these components are orthogonal to each other, which makes the whole system easy to maintain. Given a database-result pair, the *Candidate Generator module* first generates a set of candidate queries $Q_1, \cdots, Q_n$, where their query results $Q_1(D) = \cdots = Q_n(D) = R$. To distinguish the user's intended query from other candidate queries, the *Database Generator module* modifies D to D', such that D' partition queries into different groups by generating new database-result pairs for the user to examine. The *Result Feedback module* highlights the changes between the initial database-result pair and the new database-result pairs. If the user's feedback select the group containing more than one queries, the user's

feedback is returned to the *Database Generator module* for another iteration. The process terminates once QFE has identified the intended query, or none of the candidate queries are selected.

It is clear that QFE can enhance database usability. First of all, QFE can help users construct queries if they are aware of the result, but not aware of how to derive them. For example, many database users use spreadsheets or other files to store their query results and share them with one another without any annotations. It is difficult for the others to discover the query and explore the data characteristics. Our proposed approach should be helpful for users who are not familiar with SQL, and that the required input of a single example database-result pair is a reasonable requirement for users. Another feature of QFE is that it adopts an iterative data-driven approach. We believe that showing data and changes to the user can be an intuitive way to help him/her understand the essence of the query. Moreover, QFE provides friendly and efficient interactions with the user. QFE minimizes the information shown to reduce the user's effort, and the user can give feedback in time to help QFE adjust the modify strategy for the following iterations.

Besides constructing queries for users directly, QFE can also collaborate with other tools to help analyze data. For example, Howe et al. [35] have developed an ad hoc database management system called SQLShare to help users explore data. It adopts the term *starter query* to refer to a database-specific example query to help users start their analysis work [35]. These starter queries are derived from a set of tables just by analyzing their statistical properties without users' input. Without concerns of the user's real demand, these starter queries may not be helpful for the analysis purpose. However, if the user browses the data and can provide some information about the results he/she expects, then with QFE he/she can get a starter query more specifically with concern of the user's real demand. In this way, we can avoid the cost for the user to derive his or her query by trial and error, and analyze the data efficiently.

1.2 Query-based Approach

In this section, we introduce our first approach of QFE termed *Query-based approach* (Q-QFE). As shown in Figure 1.1, given a database-result pair (D, R), the *Candidat Generator module* first generates a set of candidate queries that can derive R from database D. The *Database Generator module* takes an initial database-result pair (D, R) and a set of candidate queries QC as input, and generates a new database D' to distinguish the queries in QC. Although queries in QC can generate the same result on database D, once D is updated in future, the query results may not be same any more. Here is an example.

Example 1.1. *Consider the relation Employee(Eid, name, gender, department, salary) in a company database D and the user's intended query result R, as shown in Figure 1.2.*

Eid	*name*	*gender*	*dept*	*salary*
1	Alice	F	Sales	3700
2	Bob	M	IT	4200
3	Celina	F	Service	3000
4	Darren	M	IT	5000

Database D

name
Bob
Darren

Result R

Figure 1.2: Employee database and result pair

For simplicity, assume that there are three candidate queries in QC.

Q_1: *SELECT name FROM Employee WHERE* $gender = `M'$;

Q_2: *SELECT name FROM Employee WHERE* $salary > 4000$;

Q_3: *SELECT name FROM Employee WHERE* $department = `IT'$;

Although they all have the same query results, it is obvious that they have different query semantics. If the company hires a female employee in department IT, or raises Alice's salary up to 4000, these queries will show different query results.

It is well known that if two queries Q_1 and Q_2 are not equivalent, then there exists a database D such that $Q_1(D) \neq Q_2(D)$. Based on this statement, a straightforward thought

to distinguish two queries is to generate a new database (synthetic data) that provides different query results for different queries. However, it takes more effort for users to examine an unfamiliar database and identify the correct query result. Hence, modifying the existing database to distinguish the candidate queries is a more reasonable option.

Example 1.2. *To illustrate our approach, we continue from Example 1.1. To help identify the user's target query among these three candidates, our approach will first present to the user a modified database* D_1[1] *and two possible query results,* R_1 *and* R_2 *on* D_1 *(shown in Figure 1.3):*

Employee

Eid	*name*	*gender*	*dept*	*salary*
1	Alice	F	Sales	3700
2	Bob	M	IT	3900
3	Celina	F	Service	3000
4	Darren	M	IT	5000

Database D_1

name
Bob
Darren

Result R_1

name
Darren

Result R_2

Figure 1.3: Employee database and result pair

Essentially, the modified database D_1 *serves to partition* QC *into multiple subsets. In this example,* QC *is partitioned into two subsets with the queries in* $\{Q_1, Q_3\}$ *producing the same result* R_1 *on* D_1 *and the only query in* $\{Q_2\}$ *producing the result* R_2 *on* D_1*. The user is then prompted to provide feedback on which of* R_1 *and* R_2 *is the result of her target query* Q *on* D_1*. If the user chooses* R_2*, then we conclude that the target query is* Q_2*; otherwise,* $Q \in \{Q_1, Q_3\}$ *and the feedback process will iterate with another round and present the user with another modified database* D_2 *and two possible results,* R_3 *and* R_4 *on* D_2 *(shown in Figure 1.4).*

If the user feed back that R_3 *is the result of* Q *on* D_2*, then we conclude that* Q *is* Q_1*; otherwise, we conclude that* Q *is* Q_3*. For this example, the target query is determined*

[1] The modification(s) in the database (i.e., Bob's salary) are shown as boxed text.

Employee

Eid	*name*	*gender*	*dept*	*salary*
1	Alice	F	Sales	3700
2	Bob	M	Service	4200
3	Celina	F	Service	3000
4	Darren	M	IT	5000

Database D_2

name
Bob
Darren

Result R_3

name
Darren

Result R_4

Figure 1.4: Employee database and result pair

with at most two rounds of user feedback, each of which involves a single tuple changed in the database. □

In this thesis, we propose Q-QFE, an iterative data-driven approach to distinguish a set of candidate queries, by modifying the existing database to show different query results. There could be multiple ways to modify database to partition queries. We aim to choose the modifications which minimize the user's effort as he/she examines the new database-result pairs. We present a cost model to quantify the user's effort to determine the target query relative to a modified database D', and we also demonstrate the effectiveness and efficiency of our approach using real data sets. So far, Q-QFE supports select-project-join (SPJ) queries with disjunction predicates.

1.3 Schema-based Approach

In the previous section, we introduced *Query-based approach* of QFE. The *Candidate Generator module* first generates a set of candidate queries which can derive R from database D, and then the *Database Generator module* distinguishes these queries to find the target one. There are several existing works can be used as query reverse engines for the *Candidate Generator module* [64, 70, 61]. However, these works are designed for a more general scenario, not tailored for QFE specially, the queries they generated may

not be suitable for QFE. Some of them generate too many queries to increase the user's workload, and some of them do not support selection predicates.

Query by Output (QBO) [64] is the first data-driven approach that aims to augment query results with interesting query-based characterizations of the tuples in the query result. The main idea of QBO is to get the queries to enhance the database usability including data analysis, data security, and etc. Hence, it will generate different queries in different query schemas to provide more useful information. Other works, such as [70] and [61], focus only on deriving a set of join queries without selection conditions, which narrows the query types. One main problem of these works is that the output candidates may involve too many queries, which have to be eliminated in the *Database Generator module*. It adds more burden to the *Database Generator*, and more workload for users. As there should only be one query satisfying users' query intention, the incorrect queries should be filtered as soon as possible.

The main reason that there may be too many candidate queries generated is that too many join schemas can derive different queries. Here is an example.

Example 1.3. *Consider the IMDB database with the following tables, ACTOR (pid, fname, lname, gender), MOVIE (mid, name, year), DIRECTORS (did, fname, lname), CASTS (pid, mid, role) and MOVIE_DIRECTORS (did, mid). Suppose a user needs to find the query whose result is "Fight Club". There are so many different ways to get the same answer. We can compose a query to find the only movie David Fincher directed in 1999, or the only movie Edward Norton and Brad Pitt starred together, or the only movie David Fincher and Edward Norton worked together. These three queries join different tables together and have selections on different attributes.*

To avoid generating too many candidate queries, in this section, we introduce a second approach termed *Schema-based approach* (S-QFE). In S-QFE, the *Candidate Generator module* generates a set of candidate *query schemas* instead of queries. A query schema

Employee

name	*gender*	*dept*	*salary*
Alice	F	Sales	3700
Bob	M	IT	4200
Celina	F	Service	3000
Darren	M	IT	5000
Elly	F	Seales	4300
Frank	M	Service	3700
Grace	F	IT	4000

name
Bob
Darren

Database D Result R

Figure 1.5: Employee database and result pair

contains a query's join relations, join predicates, projection attributes and selection predicate attributes. Thus, each query schema can be considered as a set of queries. With the candidate queries, the *Database Generator module* modifies the database and shows the user the differences among the candidate query schemas by the database-result examples. Similar to Q-QFE, the user examines the examples and selects the correct query schema. Then we continue to generate queries with the correct query schema.

Example 1.4. *Here is an example to illustrate S-QFE. Consider a database-result pair* (D, R)*, where* D *is a single relation with 4 attributes as shown in Figure 1.5.*

There are three candidate query schemas, namely, with selection attributes given by $\{gender, dept\}$*,* $\{gender, salary\}$ *and* $\{dept, salary\}$*. The corresponding candidate queries are shown as follows.*

Q_1*: SELECT name FROM Employee WHERE* $gender = `M'$ *AND* $dept = `IT'$*;*

Q_2*: SELECT name FROM Employee WHERE* $gender = `M'$ *AND* $salary > 4000$*;*

Q_3*: SELECT name FROM Employee WHERE* $dept = `IT'$ *AND* $salary > 4000$*;*

The query schema with only one attribute is not a candidate, because it can not generate a query Q *such that* $Q(D) = R$*.*

Now let us consider attribute $dept$ *at first, we present the user with a modified database* D_1 *and two possible query results,* R_1 *and* R_2*, on* D_1 *as shown in Figure 1.6. We modify*

Employee

name	*gender*	*dept*	*salary*
Alice	F	Sales	3700
Bob	M	Service	4200
Celina	F	Service	3000
Darren	M	IT	5000
Elly	F	Seales	4300
Frank	M	Service	3700
Grace	F	IT	4000

Database D_1

name
Bob
Darren

Result R_1

name
Darren

Result R_2

Figure 1.6: Employee database and result pair

Bob's department from "IT to "Service". If the target query schema does not have dept *as a selection attribute, then the query result should not be affected, i.e.* R_1*. Otherwise, the result should be* R_2*. The user is then prompted to provide feedback on which of* R_1 *and* R_2 *is the result of the target query on* D_1*. Based on the user's feedback, we can determine the correct query schema, and continue to generate the target query with the query schema.*

In this thesis, we propose an iterative data-driven approach to identify the target query schema and construct the target query. There are mainly two challenges. The first challenge is how to generate candidate query schemas, and the second challenge is how to modify the database to show the differences among different query schemas. So far, S-QFE only supports select-project-join queries (SPJ queries) without disjunction predicates.

1.4 Thesis Contributions

In this thesis, we make the following key contributions.

First, we propose a novel paradigm, Query From Examples, to help non-expert database users to construct queries. For users who are not familiar with SQL queries, our approach offers both an easy-to-use specification of their target queries (via a database-result pair)

as well as a low-effort mode of user interaction (via feedback on modified database-result pairs).

Second, we design a Query-based approach of QFE, which can help users to distinguish a set of queries and identify the target query.

Third, we design a Schema-based approach of QFE to identify the target query schema first and then identify the target query.

Fourth, we demonstrate the effectiveness and efficiency of our approaches using three different datasets. The first is a real dataset from SQLShare [36], a cloud-based platform designed to help scientists utilize RDBMS technology for data analysis. The second real dataset is the baseball database containing various statistics (e.g., batting, pitching, and fielding) for Major League Baseball[2], and the third one is the Adult data set extracted from the 1994 Census database[3], from the UCI Machine Learning Repository, which is a single-relation data set that has been used in many classification works.

1.5 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 presents the related work on query construction, example-driven system, query generator and data generator.
- Chapter 3 presents the *Query-based approach*. We describe the challenges and propose our algorithms to solve the problem. We also conduct an experimental study over real datasets.

[2]http://www.seanlahman.com/baseball-archive/statistics
[3]http://archive.ics.uci.edu/ml/datasets/Adult

- Chapter 4 presents the *Schema-based approach*. We propose a novel algorithm to generate candidate query schema and construct the target query. We also conduct an experimental study over real datasets.
- Chapter 5 concludes the thesis and discusses some interesting directions that future studies can undertake.

CHAPTER 2

LITERATURE REVIEW

In this chapter, we conduct a literature review over the related work of QFE. Although the title of our work is similar to *Query by Example*(QBE) [71], the problem addressed by QBE, which focuses on providing a more intuitive form-based interface for database querying, is completely different from our work. Besides, there is another work by Davide et al. [53] which shares a similar idea of QBE. The user provides a sample of example of what he needs, and the system returns the relevant answers, which might be expected by the user. Although these works use examples as ours, the problems we solve are completely different.

We classify the related works in terms of their similarities/differences with QFE. First we survey the existing works of other tools that can help users construct queries. Then we discuss the related works using example-driven methods. After that, we narrow the scope in the context of query generator. At last, the related works of database generator are reviewed.

2.1 Query Construction

It has been asserted that the database usability [39] is as important as its capability. Several different approaches have been developed with the broad objective of helping database users construct queries. These approaches differ mainly in their assumptions about the users' level of database expertise (e.g., whether users are knowledgeable in SQL), users' familiarity with the database schema, the type of help provided (e.g., query recommendation, query completion), and the available resources to help with query construction process (e.g., whether query logs of past queries are available).

One category is *query recommendation systems* [49, 11, 32, 30, 15, 5]. Query logs have been widely used for query recommendation, since they are considered as a rich source of knowledge on user behaviors. The system analyzes query logs and extracts useful queries to recommend to users. Some of these works [49, 11, 32] are implemented in search engines to provide better user experience to recommend relevant queries. They use techniques in keyword search to explore query logs, rank the suggested queries and present them to users. Some other works [30, 15, 5] monitor the current user's behavior, like keyword match, and compare it with the previous users' by looking through the query logs. If the system determines that current user has similar information need, it will suggest the queries from previous users. Since such solutions are based on the user's previous actions, and not on the user's query intention, the usefulness of the recommended queries is quite limited. Besides, they are not helpful if the user needs a new query which is not stored in query logs.

Another direction studied is *query auto-completion* [41, 55] that aims to interactively help users to compose their queries. As the user types an attribute or table name, the system will automatically provide several available query fragments like selection or join predicates on the fly. These works study the database schema or query logs, and find the most frequently used fragments, and the related tables. Then user will continue to

compose the query based on these query fragments. Some other works use the keyword search techniques to help users construct queries [63, 23, 24]. Once a user types in some keywords, the system interprets them first, and then constructs queries based on these candidate interpretations. Although these works enhance the database usability and help users to construct queries, the problem they solve is different from ours. We use query result to indicate user's query intention, and take it as the key input in our approach. All these works are based on the users' previous actions, and not on the users' query intention.

Another approach that have been proposed is *query reuse systems* [42, 35]. The idea here is to store the user's previous queries in a shared repository so that he/she (or other users) could later browse them when constructing new queries. Our QFE approach differs from all these approaches as it does not require users to be familiar with SQL and also does not rely on the availability of query logs to construct queries.

Besides the above works, Abouzied et al. proposed DataPlay [3, 4], a visualization tool to help users construct quantified queries using a trial-and-error approach. After a user provides quantified constraints to the system, the system will generate the query results for the user to examine and continue tuning and auto-correcting the incorrect query based on the user's feedback. It ranks the query correction suggestions and shows the user the effects of between the suggested queries and current incorrect query. Our work differs from their works because instead of query constraints, we ask users to provide input/output examples at the beginning. Besides, instead of refining query, our approach focuses on filtering false positive queries having the same query results on an input database, which they do not.

In addition, some researchers focus on helping users interpret queries. In [38, 62, 44], Ioannidis et al. proposed a method to explain queries using natural language. They use a graph-based model to represent a query, and then traverse the graph and compose query descriptions in natural language. Besides NL query interfaces, Gatterbauer and Danaparamita [29, 21] presented a novel system QueryViz to visualize SQL query. They take

an existing SQL query and creates a graph that helps user understand its meaning. Another approach is to use data examples to illustrate the semantics of queries [47, 56, 57]. They generate input data examples and push them into the query plan tree to get the output data. For each operator, they show intermediate data and let users understand the actual utility of each operator. Since the main focus of our approach is to show the differences of queries through database-result pairs, these works are quite different from ours.

2.2 Example-Driven Systems

The broad idea of an example-driven approach for problem solving has been applied in many diverse contexts (e.g., [7, 6, 25, 58, 69]). In [25], an interactive, example-driven approach was developed to help users explore their databases, which is related to the general framework for an automatic navigation of databases first introduced in [12]. The approach in [25] helps users to formulate a plausible SQL query based on the user's feedback on samples of database tuples presented to the user. At each iteration, the system presents the user with a sample of tuples for feedback on which of the shown tuples are relevant to the user's intention. Based on the user's feedback, the system generates a different sample of database tuples for the next iteration of user feedback. When the user decides to terminate this steering process after some number of iterations, a SQL query representing the user's intended query is generated from a classification model constructed by the system. The approach is designed to minimize the size of the samples shown and the total processing time. Our work is different from [25] in three key aspects. First, our context is different from theirs as our work is not focused on data exploration, and users using QFE are required to provide an input/output example to indicate the query intention. Second, our approach is different from theirs as QFE operates by first generating a set of candidate queries and then pruning away false positives via user feedback on several query results shown in each iteration. In addition, QFE also generates a modified database in each it-

eration to distinguish different subsets of candidate queries. In contrast, [25] generates a plausible query (out of possibly many candidate queries) using classification techniques, and their focus is not on distinguishing the candidate queries. Third, [25] supports only select-project-join queries on a single relation whereas our approach is more general.

Example-driven techniques have also been applied for debugging scheme mappings [7, 6]. In [7, 6], users are shown examples to differentiate alternative mapping specifications and find the desired mapping based on the user's interests of these data examples. Although we also show different query outputs to help the user to pick the correct query from the candidate queries, the methods are different. Unlike schema mapping, we need to modify the database to distinguish the false positive queries. Qian et al. also proposed a system for sample-driven schema mapping [58]. The user gives example tuples in a result table (or partial tuples), and the system attempts to find the best queries that will produce (at least) those results. However, they look only at project-join mappings and do not handle queries with selection.

For non-database related applications, S. Gulwani and his colleagues have developed example-driven techniques to solve many diverse problems. For instance, they have applied example-driven techniques to reformat text documents [69]. They asked user to provide input/output examples to show his intent, and reformat the source structured and semi-structured text as required. Due to the different contexts, the techniques developed there are not applicable to our work.

2.3 Query Generators

In this section, we review the related works of query reverse problem, i.e., given a database D and result R, the query reverse engine generates a query Q such that Q's result on D is R, which is also the problem *Candidate Generator module* focuses on.

Given a database D and query result R, QBO [64, 65] generates a set of candidate queries $\{Q\}$, where $Q(D) = R$. The system can also rank queries, and display the top k queries to the user to select. In [70], Zhang et al. also proposed a query reverse engine which can derive a set of join queries without selection conditions. Both works can generate a set of queries that have the same query result as R. However, their main focus is not help users construct the intended query.

[59] introduced View Definition Problem(VDP), which is to derive a view definition Q when given an input database D and a materialized view V. However, it focuses on a basic scenario where D consists of only one single relation R and the derivation of Q is essentially finding the selection predicate on R to generate V. Therefore, it cannot be extended to our case.

In [61], Shen et al. also proposed an algorithm to discover project join queries by given example tuples. Unlike QBO and QFE, the output of these join queries are not exactly the same as the given examples. The generated queries are minimal project join queries whose output contain all the tuples in given examples. Psallidas et al. [2] proposed a candidate-enumeration and evaluation framework for discovering project-join queries. Their system handles only text columns and establishes a query relevance score based evaluation of candidate queries. The system returns the PJ queries with the top-k highest scores and it discovers not only the queries that exactly match the given example tuples. As the main focus is finding join queries to cover examples, their approach is orthogonal to our problem.

Another related area is intensional query answering or cooperative answering, where for a given query Q, the goal is to augment the query's answer $Q(D)$ with additional intensional information in the form of a semantically equivalent query that is generated through the database integrity constraints [28, 52]. Two queries are semantically equivalent if for every valid database, their query results are same. If their results are same only on the

given database D, they are instance equivalent on D. It is obvious that semantic equivalence is data-independent, which is much stronger than instance equivalence, and can only be computed using database integrity constraints. In our approach, we adopt instance equivalence instead of semantically equivalent query for the following reasons. First of all, sometimes the data semantics are not explicitly captured using integrity constraints in the database for various reasons [31]. The effectiveness of intensional query could be very limited. Second, it can be very hard to derive semantically equivalent queries for complex queries. Third, intensional query answering requires the input query Q to be known, which QFE does not need. Finally, our approach focuses more on helping user construct query. Using instance equivalent queries can capture more queries with different semantics, giving us a larger chance to include user's intended query. If we generate semantically equivalent query, then we do not have this opportunity to find other queries with different semantics.

In another set of related work, Bruno et al. [10] and Mishra et al. [51] examined the problem of Targeted Query Generation (TQGen) that aims to generate test queries to meet certain cardinality constraints. TQGen takes as input a query Q, a database D, and a set of target cardinality constraints on intermediate subexpressions in Q's evaluation plan. TQGen will modify Q (by modifying the constant values in Q's selection predicates) to generate a new query Q' such that the evaluation plan of Q' on D satisfies the cardinality constraints. Different from the TQGen problem, our work aims to generate instance-equivalent queries that satisfy the content constraint of the query result. In addition, TQGen requires the input query Q to be known whereas we allow the input query to be unknown.

2.4 Database Generators

Our database generator generates a new database to distinguish the candidate queries by different query results. There are many related works, and in this section, we classify them into different classes and review them in details.

2.4.1 Reverse Query Processing

One related area is called reverse query processing [9, 8, 10, 46, 51]. Instead of generating queries, reverse query processing is to generate a database D when given a query Q and a desired query result R such that $Q(D) = R$ [9]. Reverse Query Processing (RQP) is based on a reverse relational algebra (RRA). For each operator of the relational algebra, Binnig et al. defined a corresponding operator of the reverse relational algebra that implements its reverse function. All reverse algebra operators respect the integrity constraints of the database schema in order to generate correct output. The whole data processing is started by scanning the query result and pushing each tuple down to the leaves (i.e. the base tables) of the query tree. RQP can generate synthetic data examples and be applied to some applications for verification and query debugging. That is related to some of our motivation, but at the same time, the main focus is still different.

QAGen [8] is another query-aware data generator system. It takes the query and the set of constraints (usually cardinality and data distribution) defined on the query as input, and generates a query-aware test database as output. To process a query before the data is generated, QAGen introduces the concept of symbolic query processing (SQP). QAGen uses SQP to populate a symbolic database according to the constraints and schema, and finally instantiates the symbolic tuples with a data instantiator. [46] extends it to study the generation of workload-aware data.

2.4.2 Query Equivalence Problem

Since our goal is to partition queries into different groups and show user the differences among the queries, one related research area is query equivalence or query containment problem. It has been studied extensively, since it is a fundamental problem in database research. So far, most of the existing research works focus on characterizing the query equivalence problem. They study the complexity and sufficient conditions of the query equivalence problem under different semantics (set, bag, bag-set) [18, 22, 16] and different constraints (inequality, aggregation, nested, etc) [67, 40, 22, 19, 20]. The core idea of these works to solve query containment problem is to check whether homomorphism between two queries exists. Given two queries Q_1 and Q_2, if there exists a homomorphism from query Q_1 to Q_2, then Q_2 is contained in Q_1. If Q_1 is contained in Q_2 at the same time, then two queries are equivalent. This method can help user check query containment, but it is not helpful to comprehend the differences between queries. As these works can not tell more information about the query semantics or correctness, they can not help user identify the intended query.

Another approach to check query containment is using an instance-based method [45, 66, 68, 27]. Levy and Sagiv [45] first proposed a method to generate canonical databases to test queries, which is described in [66] as well. The idea is to build an exponential number of canonical databases, and apply given queries on these databases. If there is no counterexample to the containment, then the query containment statement is true. In [68], Wei et al. gave an apriori-like algorithm to optimize the algorithm. Sharing the same principle, Farré et al. [27] presented the Constructive Query Containment (CQC) method to check query containment, which aims to construct a counterexample that proves that the query containment relationship being checked does not hold. Different from the query equivalence problem, our goal is to help users pick the correct query from a set of queries. Not just take more than two queries as input, we also avoid using synthetic data to make the examination process easier.

In [47], Mannila and Räihä first introduced a method to distinguish one query Q from a set of queries $\mathcal{Q}$. Related to the well-known concept of *Armstrong database* [26], they define the notion of complete test databases for a given query Q. The complete test database for Q is to show the non-equivalence of Q and Q_i, for every $Q_i \in \mathcal{Q}$. They further proposed a method to construct such complete test databases for Q, if it exists. However, there are several limitations about their method. First, queries with disjunctions are not supported. Besides, each query Q_i is formed from Q by removing some conditions.

In [60], Shah et al. addressed the problem of test data generation for checking correctness of SQL queries, based on the query mutation approach for modeling errors. Given a query, they generated test data to kill the query mutations. The mutant queries are pre-defined using certain query templates, such as join/outerjoin mutation (e.g., change equijoin to outer join), comparison operator mutants (e.g., change $<$ to $\leq$), and aggregation mutation, etc. A mutant query is said to be killed by a test case when the execution of the mutant query on a test case produces a different result than the execution of the original query. For example, if a query uses innerjoin ($\bowtie$) instead of left outerjoin ($⟕$) by mistake, then some result might be missing in the final result. The goal of [60] is to generate a complete data set that covers all kinds of mutations. Shah et al. proved that the decision version of the test data generation problem is NP-Hard in the size of the query, and sketched an approach to generate test data based on some assumptions, e.g., no nested queries.

The common idea of the above works is to generate database to test query equivalence, which is not an ideal method for our problem. Considering the requirement that users should be able to understand the new database easily, we hope to modify the existing database to distinguish queries, and limit the modification as few as possible.

2.4.3 Explaining Missing Answers

Recently there have been some works using the instance-based approach to explain missing answers (or why not answers [13]) [34, 37, 33]. Given an input database D, query Q and a set of missing answers T, Huang et al. [37] explained the missing answers T by modifying some tuples in the database D such that the result of the query Q on the modified database will include both the original result and the specified missing tuples T. They computed the provenance of T, which consists of the tuples T can potentially be derived from. This explanation model is very flexible if arbitrary modifications to the database are allowed to derive the missing tuples. Similarly, for missing answers T, Herschel et al. [34, 33] altered current database D to D' and got the new result $Q(D')$ that $Q(D') = Q(D) \cup T$, and each set of tuples, from where T can be derived, is called an explanation. They used the notion of homomorphism to minimize the number of explanations and showed that determining the minimal explanations for unions of conjunctive queries is $\mathcal{NP}$-*complete*. Our approach shares the same core principle of modifying database, but the problem objectives and techniques are different. Instead of the specific result $Q(D')$ containing missing answers t, our purpose is to generate a database D', which will derive different results for a set of input queries. This characteristic makes it non-trivial to extend existing algorithms for our problem. Besides, we also need to make sure that the collection of these modification is as small as possible, which is also non-trivial.

2.5 Query Refinement Problem

There is also some related work on query refinement to modify an input query so that its query result can satisfy some cardinality constraints [43]. The works in [43, 54] relax the queries that return empty result so that the modified queries will yield some answers.

As the goal there is to refine the query to return any non-empty result, the techniques there cannot be applied to our problem, which has stronger constraints to satisfy. Another related direction in [50, 17] deals with the problem when a query returns too many/few answers by refining the query to satisfy some constraints on the query result size. Similar to the work in [43], the focus there is on the size of the output but not on the content of the output, which we have to deal with in this context.

CHAPTER 3

QUERY-BASED APPROACH

In this chapter, we present our Query-based approach of QFE (Q-QFE). We first present the overview of Q-QFE approach in Section 3.1, and discuss the details in Sections 3.2 to 3.4. Section 3.5 presents additional extensions for our approach. An experimental evaluation of Q-QFE is presented in Section 3.6. Finally, we conclude in Section 3.7. The notations used in this chapter is shown in Table 3.1.

3.1 Approach Overview

To help non-expert database users construct queries, we propose a novel approach Q-QFE which takes a database-result pair (D, R) as input, and output the target query for users. Figure 3.1 illustrates the overall architecture of our approach. Note that the *Candidate*

Notation	Description
Q	Query
D	Database
D'	Modified database
R	Query result
$Q(D)$	Query Q's result on database D
A	Attribute
QC	Set of candidate queries
$balance(D)$	Balance score of database D
$minEdit(D, D')$	Minimal edit distance from dataset D to D'
$J(D)/J$	Result of joining all the join relations in query
TC	Tuple class
STC	Source-tuple-class
DTC	Destination-tuple-class

Table 3.1: Notation table of Chapter 3

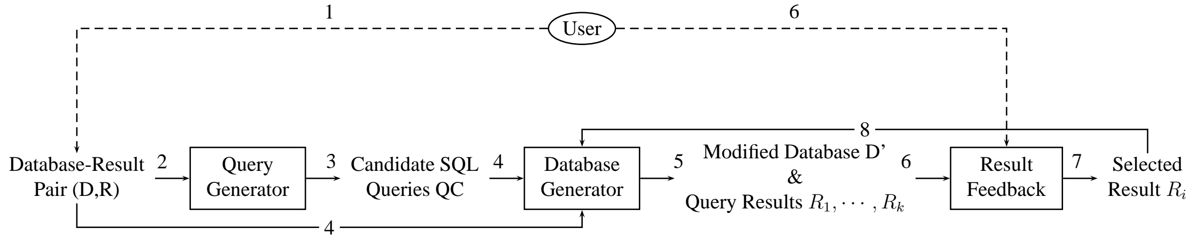


Figure 3.1: Overall Architecture of QFE

Generator module in Figure 1.1 is specialized as *Query Generator*. Q-QFE first obtains an initial database-result pair (D, R) from the user where R is the result of the user's target query on the database D. The *Query Generator* module takes (D, R) as input to generate a set of candidate SQL queries $QC = \{Q_1, \cdots, Q_n\}$ for (D, R); i.e., $Q_i(D) = R$ for each $Q_i \in QC$.

To efficiently identify the user's target query from QC, which is generally a very large collection, Q-QFE winnows this collection iteratively using a divide-and-conquer strategy. At each iteration, the *Database Generator* module takes as inputs (D, R) and $QC' \subseteq QC$, which is the set of remaining candidate queries at the start of the iteration, to generate a new database D'. The purpose of D' is to distinguish the queries in QC' based

on their query results on D'. Specifically, D' partitions QC' into a number of subsets, $QC'_1, \cdots, QC'_k$, $k \geq 1$, where two queries belong to the same subset QC'_j if and only if they produce the same result (denoted by R_j) on D'.

Next, the *Result Feedback* module presents the user with the new database D' and the collection of query results $R_1, \cdots, R_k$. If the user identifies R_x as the correct query result on D', it means that the user's target query is guaranteed to be not in QC'_j, $j \neq x$; therefore, these query subsets can be pruned from further consideration. Q-QFE will start another iteration using the subset of candidate queries QC'_x corresponding to R_x if QC'_x contains more than one query; otherwise, Q-QFE terminates with the only query in QC'_x as the user's target query.

To help reduce the user's effort to identify R_x relative to D', instead of presenting the user with a new database D' and query results $R_1, \cdots, R_k$, the *Result Feedback module* actually presents D' and R_i in terms of their differences from the original database-result pair (D, R), which is denoted by $\Delta(D, R_i)$ in Figure 3.1.

Algorithm 3.1: Q-QFE

```
   Input: A database-result pair (D, R)
   Output: Target query
1  QC = Query-Generator(D, R)
2  repeat
3  |  D' = Database-Generator(D, QC)
4  |  QC = QC_1 ∪ ··· ∪ QC_k   // Partition QC using D'
5  |  for i = 1 to k do
6  |  |  let R_i be the output of query in QC_i on D'
7  |  x = Result-Feedback(D', R_1, ···, R_k)
8  |  QC = QC_x
9  until |QC| = 1
10 return Q where QC = {Q}
```

The overall procedure for Q-QFE is shown in Algorithm 3.1. In the event that none of the query results presented at an iteration is the intended output of the user's target query (not shown in Algorithm 3.1), it means that the target query is not in the initial set of candidate queries QC. In this case, Q-QFE will initiate another round of candidate-query

generation by taking into account the information gathered to output additional candidate queries for iterative pruning.

There are two main challenges for the Q-QFE approach. The first challenge is how to generate candidate target queries given an initial database-result pair; and the second challenge is how to optimize the user feedback interactions to minimize the user's effort to identify the desired query. In this chapter, our focus is on the second challenge as existing techniques [64, 70] are available to address the first challenge.

For the Q-QFE approach to be effective, it is important to minimize the user's total effort to obtain his or her target query. A reasonable measure of a user's effort at each iteration is the amount of work required to identify the correct query result from the collection of query results $R_1, \cdots, R_k$ relative to the new database D'. Since the user is already familiar with the initial database-result pair (D, R), the user's effort at each iteration can be reduced by minimizing the following three aspects: (1) the number of query results shown (i.e., k), (2) the differences between the initial database D and the new database D', and (3) the differences between the initial query result R and each new query result R_i.

As some of these optimization objectives conflict (e.g., minimizing k could increase the number of iterations), optimizing the choice of D' to reduce the user's effort at each iteration is a non-trivial problem. In the following sections, we first present a cost model to quantify the user's effort to determine the target query relative to a modified database D', and then present the details of the key components of Q-QFE.

3.2 Cost Model

In this section, we present a cost model to quantify the user's effort in identifying the target query from an initial set of candidate queries QC. This cost model is used by

the *Database Generator module* to select a "good" modified database D' to partition QC into multiple query subsets $\{QC_1, \cdots, QC_k\}$, whose query results $\{R_1, \cdots, R_k\}$ are then shown to the user for feedback.

To minimize the number of required iterations, the size of the query subsets (i.e., $|QC_i|$) induced by the new database D' at each iteration should ideally be balanced. Given a collection of partitioned query subsets $C = \{QC_1, \cdots, QC_k\}$ induced by D', we define the *balance score* of D', denoted by $balance(D')$, to be $\frac{\sigma}{|C|}$, where σ is the standard deviation of the set $\{|QC_1|, \cdots, |QC_k|\}$. Thus, a smaller $balance(D')$ value indicates a more desirable D' that induces a partitioning with many subsets of about the same size. Furthermore, a good balance limits the worst-case number of iterations.

The user's effort is also reduced if both the differences between the initial and modified databases as well as the differences between the initial query result R and each new query result R_i are small, since new information is minimized. We quantify the difference between two instances of a relation, T and T', by the minimum edit cost to transform T to T', denoted by $minEdit(T, T')$. We consider the following three types of edit operations:

(E1) modifying an attribute value of a tuple in T,

(E2) inserting a new tuple into T, and

(E3) deleting a tuple from T.

The edit cost of (E1) is one, and both (E2) and (E3) have edit cost equal to the arity of the relation. For convenience, we use $minEdit(D, D')$ to denote the sum of $minEdit(T, T')$ for each relation T in database D that has been modified to T' in the modified database D'.

The user's effort relative to the modified database D', denoted by $cost(D')$, is modeled as

a sum of two components:

$$cost(D') = currentCost \ + \ residualCost \tag{3.1}$$

where $currentCost$ and $residualCost$, respectively, denote the user's effort for the current iteration and the remaining iterations. The effort for the current iteration is modeled as

$$currentCost = dbCost + resultCost \tag{3.2}$$

where $dbCost$ denotes the user's effort to identify the differences between the initial database D and modified database D', and $resultCost$ denotes the user's effort to identify the differences between the initial query result R and each new query result R_i. For $dbCost$, it is reasonable to expect that more effort is required from the user if the modified tuples come from a larger number of relations. Thus, we model

$$dbCost = minEdit(D, D') + \beta \times n \tag{3.3}$$

where n denotes the number of modified relations in D' and β is a scale parameter to normalize the number of relations in terms of some number of attribute modifications. For the query result differences, we have

$$resultCost = \sum_{i=1}^{k} minEdit(R, R_i) \tag{3.4}$$

Modeling $residualCost$ is somewhat trickier as it depends on the user's feedback at each iteration. A conservative estimation of this is to assume that the user's feedback in the current iteration picks the largest query subset and for each subsequent iteration, the partitioning creates only two query subsets based on a single modified database tuple. We estimate the minimum edit cost for this single tuple modification from the average of

the current iteration's database edit costs. Hence, for each subsequent iteration, $dbCost$ is modeled as $minEdit(D, D')/\mu \ + \beta$, where μ denotes the total number of modified database tuples in the current iteration. Since there are only two query subsets in each subsequent iteration, we model $resultCost$ as twice of the current iteration's average query result edit cost; i.e., $\frac{2}{k}\sum_{i=1}^{k} minEdit(R, R_i)$.

Putting everything together, we have

$$cost(D') = minEdit(D, D') + \beta \cdot n + \sum_{i=1}^{k} minEdit(R, R_i) +$$

$$N \times (minEdit(D, D')/\mu \ + \beta + \frac{2}{k}\sum_{i=1}^{k} minEdit(R, R_i)) \tag{3.5}$$

where N is the number of remaining iterations.

To minimize the user's effort, the modified database D' used in each iteration should have a small value for $cost(D')$. Note that there is a tradeoff involved in making more database modifications: although this tends to increase the cost of the current iteration, it is likely to also increase the number of query subsets in the partition (i.e., reduce the balance score of modified database) which tends to reduce the number of required iterations and the costs of the remaining iterations.

3.2.1 Estimation of Number of Iterations

The remaining issue for the cost model concerns the estimation of the number of iterations N. One simple estimation of N is given by

$$N = log_2(\max\{|QC_1|, \cdots, |QC_k|\}) \tag{3.6}$$

which is based on two assumptions about subsequent iterations: (A1) the only available query partitionings are binary ones that partition candidate queries into two subsets, and (A2) the best partitioning that creates two balanced subsets is always available.

In the following, we discuss how to improve the accuracy of this simple estimation by exploiting additional information that would be available as part of our approach (Algorithm 3 to be presented in Section 5.2). Specifically, the improvement comes from completely or partially eliminating assumption (A2).

With assumption (A1), suppose that the most balanced partitioning P in the current iteration creates two query subsets, S_x and S_y, containing x and y queries, respectively, where $x \leq y$. As before, we always assume that the largest query subset (i.e., S_y) is chosen for the next iteration. Thus, the number of "false positive" queries eliminated by the current iteration is x. Since P is the most balanced partitioning in the current iteration, it follows that for any other binary partitioning in the current iteration, the number of false positive queries eliminated by it is at most x. With this additional knowledge about x, the following property holds for each subsequent iteration.

Lemma 3.1. *Based on assumption (A1), the number of false positive queries eliminated in each subsequent iteration is at most x, where x is the number of false positive queries eliminated by the most balanced binary partitioning in the current iteration.*

Proof. We establish the proof by contradiction. Suppose that the claim is false; i.e., in some subsequent iteration with $S' \subseteq S_y$ candidate queries, there exists a binary partitioning P' that partitions S' into two subsets of u and v queries, where $u \leq v$ and $u > x$. This implies that had we chosen P' to partition the queries in the current iteration, each of the two subsets partitioned by P' would have more than x queries, contradicting the fact that P is the most balanced partitioning in the current iteration. ☐

Based on Lemma 3.1, we refine the estimation of N as the sum of two components as

follows:

$$N = N_1 + N_2 \tag{3.7}$$

$$N_1 = \lfloor (\max\{|QC_1|, \cdots, |QC_k|\})/x \rfloor - 1 \tag{3.8}$$

$$N_2 = \lceil log_2(\max\{|QC_1|, \cdots, |QC_k|\}) - xN_1) \rceil \tag{3.9}$$

Here, x denotes the number of queries in the smaller query subset created by the most balanced binary partitioning in the current iteration. In contrast to Equation (3.6), which optimistically assumes that half the number of queries are eliminated as false positives in each iteration, N_1 denotes the number of iterations where x false positive queries (i.e., the upper bound established by Lemma 3.1) are eliminated in each iteration. At the end of N_1 iterations, the number of remaining candidate queries is at most $2x - 1$, and we fall back to applying Equation (3.6) to estimate the number of remaining iterations, which is given by N_2. In the event that no binary partitioning exists in the current iteration (i.e., x is undefined), we fall back to using Equation (3.6) for the estimation of N.

3.3 Query Generator

The objective of the *Query Generator module* is to generate a set of candidate SQL queries QC for the user's target query given an initial database-result pair (D, R).

A number of approaches have recently been proposed to reverse-engineer queries given an input database-result pair [64, 70]. In this paper, we adopted the QBO approach of Tran et al. [64] for our Query Generator module as it can support more general candidate queries, specifically, select-project-join (SPJ) queries, compared to the project-join queries (i.e., without any selection predicates) considered by Zhang et al. [70].

QBO provides several configuration parameters to control the search space for equivalent candidate queries, such as the maximum number of selection-predicate attributes,

the maximum number of joined relations, the maximum number of selection predicates in each conjunct, etc. In our experiments, we configured QBO to generate as many candidate queries as possible[1].

Each generated query is of the form $\pi_\ell(\sigma_p(J))$, where ℓ and p are the query's projection list and selection predicate, respectively. J is the foreign-key join[2] of a subset of the relations in the database D. For convenience, each selection predicate is assumed to be in disjunctive normal form; i.e., $p = p_1 \vee \cdots \vee p_m$, where each p_i is a conjunction of one or more terms and a term is a comparison between an attribute and a constant.

3.4 Database Generator

The *Database Generator module* takes as input the initial database-result pair (D, R) and a set of candidate SPJ queries QC, and generates a new database D' to be used to distinguish the queries in QC. Recall that D' is used to partition QC into subsets, $QC = QC_1 \cup \cdots \cup QC_k$, such that all the queries in each QC_i generate the same output result R_i on D', and $R_1, \cdots, R_k$ are all distinct. The goal is to determine D' such that it minimizes the user's effort to identify the target query.

Assumptions. To simplify the discussion in this section, we make two assumptions about the queries QC and one assumption on D'. First, we assume that all the queries in QC share the same join schema with $J(D)$ being the foreign-key join of all the relations in the database D, simplified as J. Thus, since R determines the projection list ℓ, all the queries in QC are essentially different selection queries on the single relation J. Second, we assume that all the queries in QC preserve duplicates (i.e., the DISTINCT keyword does not appear in any query's SELECT clause). Third, we assume that any modified

[1] In practice, it might be better to set these parameters conservatively, then relax them if more candidate queries are needed.

[2] If foreign-key constraints are not explicitly provided by the user's inputs, we can infer soft foreign-key constraints by applying known techniques (e.g., [48]).

database D' is valid (i.e., D' does not violate any integrity constraints). We discuss how to relax these assumptions in Section 3.5.

3.4.1 Tuple Classes

To facilitate reasoning about the effects of database modifications on the partitioning of queries, we introduce the concept of a tuple class.

Consider a database relation $J(A_1, \cdots, A_n)$ and a set of queries QC. For each attribute A_i in J, based on the selection predicate constants involving A_i contained in the queries in QC, we can partition the domain of A_i into a minimum collection of disjoint subsets, denoted by $\mathcal{P}_{QC}(A_i)$, such that for each subset $I \in \mathcal{P}_{QC}(A_i)$ and for each selection predicate p on A_i in QC, either every value in I satisfies p or no value in I satisfies p.

Example 3.1. *Consider a relation $J(A, B, C)$ where both A and B have numeric domains; and a set of queries $QC = \{Q_1, Q_2\}$, where $Q_1 = \sigma_{(A\leq 50)\wedge(B>60)}(J)$ and $Q_2 = \sigma_{(A\in(40,80])\wedge(B\leq 20)}(J)$. We have $\mathcal{P}_{QC}(A) = \{[-\infty, 40], (40, 50], (50, 80], (80, \infty]\}$ $\mathcal{P}_{QC}(B) = \{[-\infty, 20], (20, 60], (60, \infty]\}$, and $\mathcal{P}_{QC}(C) = \{[-\infty, \infty]\}$.* □

The next example illustrates domain partitioning for non-ordered attribute domains.

Example 3.2. *Consider a relation $J(A, B, C)$ where A is a categorical attribute with an unordered domain given by $\{a, b, c, d, e, f, g\}$. Suppose that we have a set of queries $QC = \{Q_1, Q_2\}$, where $Q_1 = \sigma_{A\in\{b,c,e\}}(J)$ and $Q_2 = \sigma_{A\in\{a,b,d,e\}}(J)$. Based on the subset of domain values that match the various subsets of selection predicates in QC, the domain of A is partitioned into 4 subsets, depending on whether the values satisfy neither, both, or exactly one of Q_1 and Q_2: $\mathcal{P}_{QC}(A) = \{\{a, d\}, \{b, e\}, \{c\}, \{f, g\}\}$.* □

Given a relation $J(A_1, \cdots, A_n)$ and a set of queries QC, a *tuple class* for J relative to QC is defined as a tuple of subsets $(I_1, \cdots, I_n)$ where each $I_j \in \mathcal{P}_{QC}(A_j)$. We say that a *tuple* $t \in J$ *belongs to a tuple class* $TC = (I_1, \cdots, I_n)$, denoted by $t \in TC$, if $t.A_j \in I_j$ for each $j \in [1, n]$.

Example 3.3. *Continuing with Example 3.1,* $TC = ((40, 50], [-\infty, 20], [-\infty, \infty])$ *is an example of a tuple class for* J*, and* $(48, 3, 25) \in TC$*.* □

By the definition of tuple class, we have the property that for every query $Q \in QC$ and for every tuple class TC for a relation J relative to QC, either every tuple in TC satisfies Q or no tuple in TC satisfies Q. In the former case, we say that TC *matches* Q.

This property of a tuple class provides a useful abstraction to reason about the effects of a database modification. Specifically, we can model a single-tuple modification in a relation J by a pair of tuple classes (s, d) of J to represent that some tuple $t \in J$, where t belongs to the tuple class s (referred to as the *source-tuple class(STC)*), is modified to another tuple t', where t' belongs to the tuple class d (referred to as the *destination-tuple class(DTC)*).

Clearly, if we generate a modified database D' by modifying a single tuple t in D to t' such that both t and t' belong to the same tuple class, then all the queries in QC would still produce the same query result on D'. Thus, for QC to be effectively partitioned by D', the (STC, DTC) pair (s, d) corresponding to a modified tuple in D' must have $s \neq d$. The following result states the maximum number of query subsets that can be partitioned by a modified database.

Lemma 3.2. *Consider a set of queries* QC *that have the same query result on a database* D*, and a new database* D' *that is obtained from* D *by modifying* n *distinct tuples in* D*.* D' *can partition* QC *into at most* 4^n *query subsets,* $QC = QC_1 \cup QC_2 \cup \cdots \cup QC_m$*,* $m \in [1, 4^n]$*, such that (1) all the queries in each* QC_i *produce the same query result on* D'*, and (2) for each pair of queries* $Q_i \in QC_i, Q_j \in QC_j, i \neq j$*,* $Q_i(D') \neq Q_j(D')$*.*

Proof. Consider the case where $n = 1$. Let D' be a modified database obtained from D by modifying a single tuple t in D to t' such that the projected attribute values of t and t' are, respectively, x and x', where $x \neq x'$. For each query $Q \in QC$, there are four possibilities for $Q(D')$: (1) $Q(D') = Q(D)$ if neither t nor t' matches Q; (2) $Q(D') = Q(D) \cup \{x'\}$, if t does not match Q but t' matches Q; (3) $Q(D') = Q(D) - \{x\}$, if t matches Q but t' does not match Q; and (4) $Q(D') = Q(D) \cup \{x'\} - \{x\}$, if both t and t' match Q. Thus, since there are only 4 potential results, QC can be partitioned into at most 4 query subsets when a single tuple is modified. It follows that the maximum number of query subsets is 4^n for n tuples modifications. □

Given a database D and set of (STC, DTC) pairs S representing modifications to D, we can generate a modified database D' from D and S as follows: for each $(s, d) \in S$, choose a tuple t in D that belongs to s and modify t to t' such that t' belongs to d. Given this, it is convenient to extend the definitions of $balance(D')$, $minEdit(D, D')$ and $cost(D')$ to sets of (STC, DTC) pairs. Specifically, if D' is a modified database that is generated from D and S as described, then we define $balance(S) = balance(D')$, $minEdit(S) = minEdit(D, D')$, and $cost(S) = cost(D')$.

3.4.2 Overview of Approach

Generating a modified database D' with a small value of $cost(D')$ is a complex problem due to the large search space of possible database modifications. In this section, we present an effective heuristic approach to compute D' by searching in the smaller domain of tuple-class pairs. Our approach first finds a set S_{opt} of (STC, DTC) pairs that minimizes $balance(S_{opt})$ and $minEdit(S_{opt})$, and then maps each tuple-class pair in S_{opt} to a concrete tuple modification to form D'.

For efficiency, our search for S_{opt} is organized iteratively in increasing cardinality of the candidate tuple-pair sets: we first consider a search space consisting of single-pair sets,

Algorithm 3.2: Database-Generator

Input: A database D, a set of candidate queries QC
Output: A modified database D'
1. SP = `Skyline-STC-DTC-Pairs`(D, QC)
2. S_{opt} = `Pick-STC-DTC-Subset`(SP, QC)
3. Let D' be a modified database generated from D and S_{opt}
4. **return** D'

and then extend this to consider a search space of two-pair sets, and so on. The search space extension from i-pair sets to $(i+1)$-pair sets is done in such a way that only "good" candidates are considered, to limit the search space.

The search space for single-pair sets is generated by considering the skyline (STC, DTC) pairs defined with respect to their balance scores and minimum edit costs. Given two (STC, DTC) pairs, (s, d) and (s', d'), we say that (s, d) *dominates* (s', d') if (1) $balance(\{(s, d)\}) \leq balance(\{(s', d')\})$, (2) $minEdit(s, d) \leq minEdit(s', d')$, and (3) at least one of the two inequalities in (1) and (2) is strict. A set S of skyline (STC, DTC) pairs has the property that for every two distinct pairs $(s, d), (s', d') \in S$, neither (s, d) nor (s', d') dominates the other.

The overall design of the database generator module is shown in Algorithm 3.2, which takes the initial database D and a set of candidate queries QC as inputs and outputs a modified database D' with a small value of $cost(D')$. The algorithm first generates a set SP of skyline (STC, DTC) pairs from D and QC using the function `Skyline-STC-DTC-Pairs`. The second step selects a "good" subset of (STC, DTC) pairs $S_{opt} \subseteq SP$ using the function `Pick-STC-DTC- Subset`. Finally, the modified database D' is generated from D and S_{opt}.

3.4.3 Algorithm Skyline-STC-DTC-Pairs

The function `Skyline-STC-DTC-Pairs`, shown in Algorithm 3.3, takes the initial database D and a set of candidate queries QC as inputs to generate a set of skyline

Algorithm 3.3: Skyline-STC-DTC-Pairs

Input: The initial database D, a set of candidate queries QC
Output: A set of skyline tuple-class pairs

```
1  STC = set of source-tuple classes derived from D & QC
2  initialize set of skyline tuple-class pairs SP = ∅
3  initialize minbalance = ∞
4  let n be the number of distinct selection-predicate attributes in QC
5  for i = 1 to n do
6  |   initialize SP_i = ∅
7  |   foreach s ∈ STC do
8  |   |   let DTC = set of destination-tuple classes that can be derived from s by modifying
   |   |   i subsets
9  |   |   foreach d ∈ DTC do
10 |   |   |   p = (s, d)
11 |   |   |   if balance({p}) < minbalance then
12 |   |   |   |   SP_i = {p}
13 |   |   |   |   minbalance = balance({p})
14 |   |   |   else if balance({p}) == minbalance then
15 |   |   |   |   SP_i = SP_i ∪ {p}
16 |   SP = SP ∪ SP_i
17 |   if the running time is larger than threshold δ then
18 |   |   break
19 return SP
```

(STC, DTC) pairs SP.

The function first generates the set of all the source-tuple classes STC from D and QC. Recall that all the queries in QC are assumed to be selection queries on a single relation J formed by joining all the relations in D based on their foreign-key relationships. The source-tuple classes are derived by first using QC to compute $\mathcal{P}_{QC}(A_i)$ for each attribute A_i in the selection predicates in QC, and then mapping each tuple in J to its source-tuple class.

The skyline (STC, DTC) pairs are generated iteratively in order of non-descending minimum edit cost starting from one to n, where n is the number of distinct attributes that appear in the selection predicates in QC. Thus, the ith iteration generates SP_i, the set of skyline (STC, DTC) pairs with a minimum edit cost of i. By enumerating the skyline pairs in this manner, any dominated tuple class pairs can be detected efficiently and pruned.

The time complexity of this function is $O(mk^n)$, where m is the total number of source-tuple classes and k is the maximum number of domain subsets over all selection-predicate attributes; i.e., $k = \max_{A_i}\{|\mathcal{P}_{QC}(A_i)|\}$. Note that in the ith iteration, the number of destination-tuple classes that can be generated from one source-tuple class is $C_i^n(k-1)^i$. Therefore, the total number of (STC, DTC) pairs considered is at most $\sum_{i=1}^{n} C_i^n(k-1)^i$, i.e., $O(k^n)$.

Given the high time complexity of this function, in our experimental evaluation, we used a threshold parameter δ to control the maximum running time allocated for this function. Once the threshold is reached, the function terminates and returns all the skyline pairs that it has enumerated so far.

3.4.4 Algorithm Pick-STC-DTC-Subset

The function `Pick-STC-DTC-Subset`, shown in Algorithm 3.4, takes as inputs the set of skyline (STC, DTC) pairs SP and the set of candidate queries QC to select a "good" subset of SP for deriving D'. Steps 1 to 8 consider the search space of single-pair sets and identify the optimal sets with minimum cost, which are maintained in L. Steps 9 to 21 consider the search space of i-pair sets iteratively, $i \in [2, |SP|]$, which is extended from the search space of $(i-1)$-pair sets, denoted by OP_{i-1}. To maintain a small search space of good candidates for the next iteration, only those i-pair sets that have a lower balance score relative to their constituent $(i-1)$-pair sets are used for the next iteration. Finally, in the event that L contains more than one optimal set, step 22 picks the optimal set with the lowest balance score. The time complexity of Algorithm 3.4 is $O(2^{|SP|})$. Although the worst-case complexity is high, our experimental results show that in practice, the size of the search space considered is small due to our balance-score-based pruning heuristic.

Algorithm 3.4: Pick-STC-DTC-Subset

Input: A set of skyline (STC, DTC) pairs SP, a set of candidate queries QC
Output: A subset of (STC, DTC) pairs $S_{opt} \subseteq SP$

```
1  initialize L = ∅
2  initialize mincost = ∞
3  foreach p ∈ SP do
4  |  if cost({p}) < mincost then
5  |  |  L = {{p}}
6  |  |  mincost = cost({p})
7  |  else if cost({p}) == mincost then
8  |  |  L = L ∪ {{p}}
9  initialize OP_1 = SP
10 for i = 2 to |SP| do
11 |  initialize OP_i = ∅
12 |  foreach op ∈ OP_{i-1} do
13 |  |  foreach p ∈ SP, p ∉ op do
14 |  |  |  op' = op ∪ {p}
15 |  |  |  if balance(op') < balance(op) then
16 |  |  |  |  OP_i = OP_i ∪ {op'}
17 |  |  |  if cost(op') < mincost then
18 |  |  |  |  L = {op'}
19 |  |  |  |  mincost = cost(op')
20 |  |  |  else if cost(op') == mincost then
21 |  |  |  |  L = L ∪ {op'}
22 let S_opt ∈ L such that balance(S_opt) ≤ balance(S) ∀ S ∈ L
23 return S_opt
```

Side Effects of Tuple-Class Modifications

Recall that given a set of (STC, DTC) pairs S, $cost(S)$ is derived by first mapping each tuple-class pair $(s, d) \in S$ to a pair of tuples (t, t'); where $t \in D$ belongs to s, and t' is modified from t such that t' belongs to d. The set of derived modified tuples form D', and $cost(S)$, which is defined to be $cost(D')$, is computed using Equation (3.5).

In general, a single database tuple modification from t to t' could result in more than one result tuple in $Q(D)$ being modified, since the modified base tuple could join with multiple tuples and therefore contribute to multiple result tuples as illustrated by the following example.

Example 3.4. *Consider the following joined relation* $J = T_1(\underline{A}, B, C) \bowtie_A T_2(\underline{A, D})$,

where $T_2.A$ is a foreign key that references T_1.

A	*B*	*C*	*D*
1	*10*	*50*	*20*
1	*10*	*50*	*40*
2	*80*	*45*	*25*
3	*92*	*80*	*20*

$J = T_1(A, B, C) \bowtie_A T_2(A, D)$

Assume that there is a (STC, DTC) pair (s, d) that corresponds to modifying the value of attribute B in the base tuple $(1, 10, 50)$ in T_1 to some other value. This single-tuple modification in T_1 actually affects the first two tuples in J. □

Thus, the database modification corresponding to a single tuple-class pair can potentially affect more than one query result tuple. Since the affected tuples might not belong to the same destination-tuple class, we need to take into account such unintended effects to accurately compute $cost(S)$.

Our implementation of Q-QFE constructs a join index for each foreign-key relationship in the database to efficiently keep track of the set of related tuples (with respect to the foreign-key relationship) for each base tuple. Using the join index, the unintended side effects of a modification corresponding to tuple-class pair can be easily identified to accurately compute the cost. The algorithm is shown in Algorithm 3.5, which takes as input a tuple-class pair (s, d) and initial database D, and outputs the cost of (s, d) and the tuple assigned for (s, d) to be modified. To minimize $resultCost$, tuple-class modifications that have no side-effects are preferred.

As shown in Algorithm 3.5, given a (STC, DTC) pair, we first map STC to all the corresponding tuples, which are managed in hash buckets previously when we compute

Algorithm 3.5: Computing Cost

Input: Tuple-class pair (s, d), database D
Output: cost c, and chosen tuple t_{min} of (s, d)

```
1  Ts = set of tuples belonging to STC s;
2  min_n = ∞, t_min = ∅, j_min = ∅;
3  foreach t ∈ Ts do
4      sum = 0, Js = ∅;
5      let T_b be the modified tuples from base relations;
6      foreach t_b ∈ T_b do
7          let J be the set of tuples in joined relation composed of t_b;
8          J = J − {t};
9          n = |J|;
10         if n > 0 then
11             sum+ = n;
12             Js = Js ∪ J
13     if sum == 0 then
14         compute c = cost(s, d) with cost model;
15         t_min = t, min_n = 0;
16         break;
17     else
18         if sum < min_n then
19             min_n = sum;
20             t_min = t;
21             j_min = Js;
22 if min_n > 0 then
23     foreach j ∈ j_min do
24         update the balance(s, d) and minEdit(s, d) with side effect of joined tuple
           j;
25     compute cost c based on updated balance(s, d) and minEdit(s, d) with cost
       model;
26 return c, t_min;
```

all the STCs(in step 1 of Algorithm 3.3). Then we examine whether side effect exists for a given (STC, DTC) pairs in steps 3 to 21. As mentioned in Section 23, we build a join index to help us detect side effect. The join index is composed of two parts: (1) for each tuple t in joined relation, the base-relation tuples which are derived from t are stored in an array; (2) for each base-relation tuple t_b, we store the tuples in join relation which are joined by t_b in an array too. With join index, we first find the modified base tuples with constant time complexity(steps 5); and for each modified base tuple, we detect the influenced tuples in join relation(step 7). If there are more joined tuples rather than

the given tuple t, we can determine that side effect exists(steps 8 to 12). Otherwise, we terminate the enumeration and choose t as the tuple to be modified later without side effect(steps 13 to 21). If we cannot find a modified tuple without side effect, we choose the tuple with minimal effected joined tuples to update the $balance(s, d)$ and $minEdit(s, d)$, and compute the cost based on our cost model(steps 22 to 25). The complexity is $O(m*n)$, where m is the number of tuples belonging to the given STC and n is the number of modified base tuples.

Note that this algorithm is only executed once for the single-pair sets in Algorithm 3.4. Afterwards, a particular tuple has been allocated to each (STC, DTC) pair for modification. When we extend the search space to i-pairs sets, the cost can be computed directly based on the cost model, without considering side effect again.

3.5 Discussion

We first discuss in Sections 3.5.1 to 3.5.3 how our approach can be generalized by relaxing the three assumptions stated in Section 3.4. We conclude with a discussion of how our approach can be extended to support more expressive queries in Section 3.5.4.

3.5.1 Queries with Set-based Semantics

So far, our discussion is based on the assumption of bag-semantics for the queries QC, where duplicate values are preserved in the query results. We now explain how our approach can handle queries with set-semantics, where there are no duplicate values in the query results.

Consider an example where the schema of $Q(D)$ consists of a single attribute A and we are trying to distinguish the set of queries $QC = \{Q_1, Q_2\}$ with an appropriate D'. There

are two basic ways to achieve this goal. The first approach is to modify D such that some value, say $a_1 \in Q(D)$, is removed from $Q_1(D')$ but remains in $Q_2(D')$. The second approach is to modify D such that some value of attribute A, say $a_2 \notin Q(D)$, is inserted into $Q_1(D')$ but is not present in $Q_2(D')$.

For the first approach, we need to modify the set of tuples $S \subseteq D$ that match Q_1 with $\pi_A(S) = \{a_1\}$ such that the modified tuples do not match Q_1. For the second approach, it is sufficient to modify a single tuple in D such that the modified tuple t has $t.A = a_2$ and t matches Q_1 but not Q_2. The first approach is more complex to handle since the set of tuples S to be modified might not all belong to the same tuple class. Thus, our existing Q-QFE solution can handle set semantics by adopting the second approach. Further research is required to incorporate the first approach as well into Q-QFE.

3.5.2 Queries with Different Join Schemas

We have so far assumed that all the queries in QC share the same join schema. Our approach can be extended quite easily to handle the more general case where this assumption does not hold.

The simplest approach to handle different join schemas is to use a divide-and-conquer strategy. We first partition QC into different groups so that queries in the same group share the same join schema and then apply Q-QFE on each of these groups. There are different strategies to order the query groups for processing. One strategy is to process the query groups in non-ascending order of the group size based on the assumption that the target query is more likely to be contained in a larger query group. Once the target query is identified in some query group, the processing terminates without the need to process the remaining query groups.

A more complex approach to solve the problem is to compute a full-outer join of all the relations in the database and to extend our existing Q-QFE approach to work with this

single joined relation. We plan to evaluate the tradeoffs of these different approaches as part of our future work.

3.5.3 Database Constraints

We have so far not discussed how to ensure that the generated modified databases are valid with respect to the database integrity constraints that could be provided by the users. For primary key constraints, it is trivial to ensure that modified tuples do not violate such constraints. For foreign key constraints, care must be taken to ensure a modified non-null foreign key value refers to an existing primary key value. However, more research is required to look into handling more complex database constraints.

3.5.4 Supporting More Expressive Queries

In this section, we discuss how our approach could be extended to handle more expressive queries.

For select-project-join-union (SPJU) queries, the problem of distinguishing two SPJU queries can be reduced to that of distinguishing two SPJ queries with some additional checking. For example, consider the problem of distinguishing two SPJU queries $Q_1 = Q_{11} \cup Q_{12}$ and $Q_2 = Q_{21} \cup Q_{22}$ with a modified database D'. Assume that t is an output tuple that is produced by both Q_{11} and Q_{21} on database D. The problem could be viewed as distinguishing two SPJ queries Q_{11} and Q_{21}. One way is to generate D' such that $t \in Q_{11}(D')$ and $t \notin Q_{21}(D')$; additionally, we need to ensure that $t \notin Q_{22}(D')$. Another way is to modify the database such that a new output tuple t' is contained in $Q_{11}(D')$ but not in $Q_2(D')$.

Supporting group-by aggregation (SPJA) queries, however, requires more significant extensions to our approach due to the larger number of diverse options to distinguish such

complex queries. We plan to investigate this issue more thoroughly as part of our future work.

3.6 Experimental Evaluation

In this section, we evaluate the efficiency and scalability of our approach using two real datasets. Our experiments were performed on a PC with an Intel Core 2 Quad 2.83GHz processor, 4GB RAM, and 256GB SATA HDD running Ubuntu Linux 12.04. The algorithms were implemented in C++ and the database was managed using MySQL Server 5.5.27. All timings reported were averaged over three runs.

The default values for the two configurable parameters in our approach are as follows: $\beta = 1$ for the scale parameter in Equation (3.3), and $\delta = 1s$ for the time threshold parameter in Algorithm 3. We examine the sensitivity of these parameters in Sections 3.6.3 and 3.6.4.

Sections 3.6.2 to 3.6.6 present experimental results where the result feedback interactions were automated without involving any real users, by always choosing the largest query subset (to examine worst-case behavior) in each feedback iteration. This practical approach enables us to conveniently conduct many experiments to evaluate the effects of different parameters on various properties of our approach, including the number of feedback iterations, the number of database and result modifications, and the execution time of the algorithms. Finally, Section 3.6.7 to Section 3.6.9 briefly report additional experimental results, including the effects of input example size, the entropy of the active domains of attribute, and a simple user study.

3.6.1 Database and Queries

Our experiments were conducted using two real datasets. The first dataset is a scientific database of biology information taken from SQLShare[3] that consists of two tables: the first table, named “PmTE_ALL_DE”, contains 3926 records with 16 attributes; and the second table, named
“table_Psemu1FL_RT_spgp_gp_ok”, contains 424 records with 3 attributes. The foreign-key join of these tables is a relation with 417 tuples. We used two actual queries (denoted as Q_1 and Q_2 below) posed by a biologist on this database.

The second dataset is a baseball database containing various statistics (e.g., batting, pitching, and fielding) for Major League Baseball[4]. In our experiments, we used only three of its tables ($Manager$, $Team$ and $Batting$) which have 11, 29, and 15 columns; and contain 200, 252, and 6977 tuples, respectively. The foreign-key join of these three tables is a relation with 8810 tuples. Four synthetic queries were used on this dataset (denoted by Q_3 to Q_6 below) with varying complexity in terms of the number of relations, and use of conjunctions and disjunctions in the selection predicates.

$$Q_1 = \pi_*(\sigma_{P.logFC_Fe<0.5 \wedge P.logFC_Fe>-0.5 \wedge P.logFC_P<-1 \wedge P.logFC_Si<-1 \wedge P.logFC_Urea<-1 \wedge (P.PValue_Fe<0.05 \vee P.PValue_P<0.05 \vee P.PValue_Si<0.05 \vee P.PValue_Urea<0.05)}) (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_2 = \pi_*(\sigma_{P.logFC_Fe<1 \wedge P.logFC_P>1 \wedge P.logFC_Si>1 \wedge P.logFC_Urea>1 \wedge (P.PValue_Fe<0.05 \vee P.PValue_P<0.05 \vee P.PValue_Si<0.05 \vee P.PValue_Urea<0.05)}) (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_3 = \pi_{managerID,year,R}(\sigma_{teamID=\text{“}CIN\text{”} \wedge year>1982 \wedge year<=1987}) (Manager \bowtie Team)$$

[3] http://escience.washington.edu/sqlshare
[4] http://www.seanlahman.com/baseball-archive/statistics

$$Q_4 = \pi_{ManagerID,year,2B}(\sigma_{playerID=\text{“}sotoma01\text{”} \vee playerID=\text{“}brownto05\text{”} \vee playerID=\text{“}pariske01\text{”} \vee playerID=\text{“}welshch01\text{”}})(Manager \bowtie Team \bowtie Batting)$$

$$Q_5 = \pi_{ManagerID,year,HR}(\sigma_{playerID=\text{“}rosepe01\text{”} \wedge HR>1 \wedge 2B<=3})(Manager \bowtie Team \bowtie Batting)$$

$$Q_6 = \pi_{ManagerID,year,3B}(\sigma_{playerID=\text{“}esaskni01\text{”} \wedge (IP>4380 \vee (IP<=4380 \wedge BBA<=485))})(Manager \bowtie Team \bowtie Batting)$$

The cardinalities of the query results for Q_1 to Q_6 are, respectively, 1, 6, 5, 14, 4, and 4 tuples. Each of the above queries Q is used to generate an initial (D, R) pair, and the target query in an experiment could be Q or one of the candidate queries generated from (D, R).

3.6.2 Results for Default Settings

In this section, we present experimental results for the default settings with $\beta = 1$ and δ = 1s, where the largest query subset is always chosen at each iteration. Here we discuss only the results for the scientific database; the results for the baseball database will be partially presented in Section 3.6.3.

Both Q_1 and Q_2 require 6 iterations of result feedback with our prototype. Table 3.2 shows the following per-round performance statistics: (1) the number of candidate queries and (2) the number of query subsets partitioned at the start of each iteration; (3) the number of skyline tuple-class pairs enumerated by Algorithm 3.3; (4) the total execution time, which is the sum of the running time for the Query Generator module (as part of the first iteration) and Database Generator module, and running time for modifying the database; (5) the database modification cost, $dbCost$; (6) the query result modification cost, $resultCost$; and (7) the average query result modification cost, $avgResultCost$, which is given by the ratio of (6) to (2).

Iteration No.	1	2	3	4	5	6
# of queries	19	15	13	11	10	8
# of query subsets	2	2	2	2	2	8
# of skyline pairs	2	100	52	101	51	98
Execution time (s)	2.84	1.91	1.71	1.89	1.91	1.99
$dbCost$	1	2	2	1	2	8
$resultCost$	12	11	12	11	13	80
$avgResultCost$	6	5.5	6	5.5	6.5	10

(a) Results for Query Q_1

Iteration No.	1	2	3	4	5	6
# of queries	19	11	7	5	3	2
# of query subsets	2	2	2	2	2	2
# of skyline pairs	50	6	63	130	54	12
Execution time (s)	2.91	1.69	1.81	2.89	0.69	0.71
$dbCost$	1	2	2	2	1	2
$resultCost$	11	9	10	11	11	12
$avgResultCost$	5.5	4.5	5	5.5	5.5	6

(b) Results for Query Q_2

Table 3.2: Per-round statistics for scientific database.

Note that the total execution times (over 6 iterations) for Q_1 and Q_2 are 11.25s and 10.11s, respectively, of which less than 1 second is spent on the Query Generator module. As expected, the first iteration took the most time as it included the query generation time and the first iteration also processed the largest set of candidate queries. Generally, the execution time decreases as the set of candidate queries progressively becomes smaller. However, for Q_2, observe that there is an increase in the execution time for its fourth iteration, which is due to the large number of skyline tuple-class pairs enumerated for that round. The maximum and average per-round execution times are about 3 and 2 seconds, respectively.

In terms of modification costs, the highest costs were incurred in the last iteration for Q_1 where the queries were partitioned into 8 subsets resulting in 8 database attributes and 7 query result tuples being modified. For each of the other iterations, the queries were partitioned into 2 query subsets requiring modifications of at most 2 database attributes

and a single query result tuple. Thus, the average modification cost for each round is low, implying that the expected user's effort to provide result feedback is modest.

Besides the worst-case result feedback simulation, we also experimented with an automated result feedback that always choose the query subset that contains the target query. For Q_1, it required 6 iterations, as with the worst-case results just presented. For Q_2, only 4 iterations were needed to determine the target query with a total running time of 7.4s and an average per-round modification cost of 1 database attribute and an average of 5 modified attributes for each query result.

3.6.3 Effect of Scale Factor β

In this section, we examine the effect of the scale parameter β on performance by varying its value in the range $\{1, 2, 3, 4, 5\}$ on the number of iterations and the actual total modification costs (i.e., for both database and query result modifications). Recall that the parameter β is used in Equation (3.3) of the cost model to normalize the number of relations in terms of number of attribute modifications.

For both queries Q_1 and Q_2 on the scientific database, neither the number of iterations nor the actual modification costs were affected by the variation in β.

The results for queries Q_3 to Q_6 on the baseball database are shown in Table 3.3. In terms of the effect on the number of iterations, only queries Q_3 and Q_4 were slightly affected with a decrement of one round when β is increased to 2 and 3, respectively. In terms of the effect on the modification costs, only Q_4's cost was affected with an increment of 3 when β is increased to 3.

Our experimental results indicate performance does not depend greatly upon β. The reason is that when the modified tuples come mostly from the same relation, the value of β

Query	Effect of β on number of iterations					Effect of β on modification cost				
	1	2	3	4	5	1	2	3	4	5
Q_3	7	6	6	6	6	29	29	29	29	29
Q_4	6	6	5	5	5	24	24	27	27	27
Q_5	7	7	7	7	7	32	32	32	32	32
Q_6	5	5	5	5	5	25	25	25	25	25

Table 3.3: Effect of β for baseball database

does not matter. For Q_1, except for the last iteration where two relations were modified, only one relation is modified in each iteration. For Q_2, only one relation is modified in all iterations. For Q_3 and Q_6, except for one iteration which modified only one relation, all iterations modified two relations. For Q_4 and Q_5, only one relation is modified in all iterations. Given this behavior, all our experiments used the default value of 1 for β.

3.6.4 Effect of Time Threshold δ

In this section, we examine the effect of the time threshold parameter δ on performance by varying δ in the range $\{0.1, 0.2, 0.5, 1, 2, 5, 10\}$.

Table 3.4 shows the effect of δ on the number of iterations, total modification cost, and execution time for the scientific database. Although the execution time generally increases with δ, an increase in δ could reduce the overall execution time. This is because by increasing the time for finding skyline tuple-class pairs (i.e., Algorithm 3), the quality of the subset of tuple-class pairs derived by Algorithm 4 could improve leading to a more balanced partitioning of the candidate queries thereby possibly reducing the number of iterations or modification cost. For example, in Table 3.4(a), the execution time for Q_1 decreases when δ increases from 0.1 to 0.2, due to a decrease in the number of iterations. Similarly in Table 3.4(b), the execution time for Q_2 decreases when δ increases from 0.1 to 0.2 for the same reason.

δ (s)	0.1	0.2	0.5	1	2	5	10
# of iterations	11	9	9	6	5	8	8
Modification cost	201	201	179	155	155	122	122
Execution time (s)	9.7	9.0	12.2	11.2	14.1	47.4	83.2

(a) Effect of δ on Q_1

δ (s)	0.1	0.2	0.5	1	2	5	10
# of iterations	7	4	6	6	4	4	4
Modification cost	87	90	74	74	70	70	70
Execution time (s)	7.2	5.1	8.1	10.0	14.4	26.3	48.4

(b) Effect of δ on Q_2

Table 3.4: Effect of δ for scientific database

For the baseball database (results not shown due to space constraints), we observe that for queries Q_3, Q_5 and Q_6, their lowest execution times occurred when $\delta = 1s$, and for Q_4, its lowest execution time occurred when $\delta = 2s$.

Our experimental results suggest that a reasonable value for the time threshold parameter is 1 or 2 seconds.

3.6.5 Efficiency of Algorithm 3.4

In this section, we examine the efficiency of Algorithm 3.4 in finding a "good" subset of tuple-class pairs to generate the modified database. Although the algorithm has a time complexity of $O(2^{|SP|})$, where SP denote the input set of skyline tuple-class pairs, our experimental results demonstrate that the algorithm actually performs well in practice even with a reasonably large input set for SP.

Table 3.5 shows performance results of Algorithm 4 for queries Q_1 and Q_2 on the scientific database. Recall that both queries require 6 iterations with the default worst-case automated result feedback. For each query, Table 3.5 shows the number of skyline tuple-class pairs (i.e, $|SP|$) and the execution time of Algorithm 4 for each iteration.

	Iteration No.	1	2	3	4	5	6
Q_1	# of skyline pairs	2	100	52	101	51	98
	Exec. time (ms)	0.0689	189	11.5	161	33.7	283
Q_2	# of skyline pairs	50	6	63	130	54	12
	Exec. time (ms)	125	0.598	131	1267	7.71	1.78

Table 3.5: Performance of Algorithm 4 for scientific database

The results show that the running times of Algorithm 3.4 were very short. For Q_1, the longest running time was 0.283 seconds in last iteration; and for Q_2, the longest running time was slightly over one second in the 4th iteration.

To evaluate the scalability of Algorithm 4 with respect to $|SP|$, we consider the 2nd iteration for Q_1 with $|SP| = 100$ which was generated with $\delta = 1s$. By progressively increasing the time threshold to 15 seconds, we generated 5 subsets of skyline tuple-class pairs of increasing size with $|SP| \in \{200, 400, 600, 800, 1000\}$. Table 3.6 compares the execution timings of Algorithm 4 for these 5 subsets. We also show the maximal number of reduced candidate pair sets in one iteration of Algorithm 4.

# of skyline pairs	200	400	600	800	1000
Exec. time (s)	3.22	24.55	65.76	104.54	156.49
Max. # of reduced sets	155	241	301	470	649

Table 3.6: Performance of Algorithm 3.4 for varying $|SP|$

The results show that the performance of Algorithm 4 was still reasonably fast (less than 25s) when $|SP| = 400$. We also observed that the query partitionings produced by Algorithm 4 were all the same as the size of the skyline tuple-class subset was increased from 50 to 1000. Thus, this suggests that the size of SP need not be large to find good query partitionings.

3.6.6 Effect of Number of Candidate Queries

In this section, we examine the effect of the number of candidate queries produced by the Query Generator module. Due to space constraints, we present the results only for Q_2.

To go beyond the 19 initial candidate queries generated for Q_2, we generated 61 additional candidate queries from the initial candidate queries by modifying their selection predicate constants. From the 80 candidate queries for Q_2, we created 6 subsets of candidate queries (denoted by $S_1, S_2, \cdots, S_6$) such that $S_1 \subset S_2 \subset \cdots \subset S_6$ and $Q_2 \in S_1$. The cardinality of these query subsets and their performance results are shown in Table 3.7.

Candidate query set	S_1	S_2	S_3	S_4	S_5	S_6
# of candidate queries	5	10	20	40	60	80
# of selection attributes	9	14	18	18	18	18
# of iterations	2	3	4	5	6	6
Execution time (s)	3.9	6.4	8.5	7.7	9.4	10.0
Modification cost	37	49	70	82	104	103
Avg. dbCost per round	1.5	2	1	1.6	1.5	2.2
Avg. resultCost per result set	6.8	6.1	6.6	6.2	6.3	6

Table 3.7: Effect of the number of candidate queries on Q_2

Note that the execution timings reported here did not include the running time of the Query Generator module, since we had manually generated additional candidate queries; and in any case, the candidate-query generation time was only a small fraction of the total execution time. Observe also that both the number of iterations and execution time increase with the number of candidate queries, and the per-round database and query result modification costs are reasonably low.

Since the first iteration's running time is the most time-consuming, Table 3.8 presents a breakdown of this running time in terms of the time spent at each of the three key steps of the Database Generator module (i.e., Algorithm 3.2).

Query set	S_1	S_2	S_3	S_4	S_5	S_6
Algorithm 3	1.04	1.12	1.10	1.10	1.10	1.10
Algorithm 4	0.11	0.0006	0.00007	0.000065	0.005	0.002
Modify DB	0.68	0.70	0.67	0.68	0.68	1.02
Total	2.94	2.88	2.85	2.86	2.89	3.24

Table 3.8: Breakdown of first iteration's runing time (in sec)

Observe that the running time is dominated by the first and third steps, with Algorithm 3.4

incurring the least amount of time. The results demonstrate that our approach can scale for a reasonably large number of candidate queries.

3.6.7 Effect of Initial Database-Result Pair

In this section, we present additional experimental results to evaluate the effect of the initial database-result pair on performance.

Figure 3.2 shows the queries used in this experiment, where Q_1 to Q_5 are on the scientific database and Q_6 to Q_9 are on the baseball database. Note that among the 5 queries on the scientific database, two of them are real queries and the remaining three are synthetic queries.

$$Q_1 = \pi_*(\sigma_{P.logFC_Fe<0.5 \wedge P.logFC_Fe>-0.5 \wedge P.logFC_P<-1 \wedge P.logFC_Si<-1 \wedge P.logFC_Urea<-1 \wedge (P.PValue_Fe<0.05 \vee P.PValue_P<0.05 \vee P.PValue_Si<0.05 \vee P.PValue_Urea<0.05)})(PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_2 = \pi_*(\sigma_{P.PValue_Si<=7.02e-06})(PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_3 = \pi_*(\sigma_{S.Groups>24 \wedge S.Groups<=27}(PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_4 = \pi_*(\sigma_{P.logCPM_P<=3.91148 \wedge P.logCPM_P>3.79204})(PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_5 = \pi_*(\sigma_{P.logFC_Fe<1 \wedge P.logFC_P>1 \wedge P.logFC_Si>1 \wedge P.logFC_Urea>1 \wedge (P.PValue_Fe<0.05 \vee P.PValue_P<0.05 \vee P.PValue_Si<0.05 \vee P.PValue_Urea<0.05)})(PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)$$

$$Q_6 = \pi_{ManagerID,year,HR}(\sigma_{playerID=\text{“}sotoma01\text{”} \vee playerID=\text{“}brownto05\text{”} \vee playerID=\text{“}pariske01\text{”} \vee playerID=\text{“}welshch01\text{”}})(Manager \bowtie Team \bowtie Batting)$$

$$Q_7 = \pi_{ManagerID,year,HR}(\sigma_{(playerID=\text{“}foleyto02\text{”} \vee playerID=\text{“}vangoda01\text{”} \vee playerID=\text{“}mcgrite01\text{”} \vee playerID=\text{“}jonestr01\text{”} \vee playerID=\text{“}housepa02\text{”}) \wedge (managerID=\text{”}rosepe01m\text{”} \vee managerID=\text{”}rappve99m\text{”} \vee managerID=\text{”}nixonru01m\text{”}) \wedge Batting.RBI>9 \wedge Batting.SB<=12})(Manager \bowtie Team \bowtie Batting)$$

$$Q_8 = \pi_{ManagerID,year,3B}(\sigma_{playerID=\text{”}esaskni01\text{”} \wedge IP<=4380 \wedge BBA<=485})(Manager \bowtie Team \bowtie Batting)$$

$$Q_9 = \pi_{managerID,year,Rank}(\sigma_{teamID=\text{“}CIN\text{”} \wedge year>1982 \wedge year<1988})(Manager \bowtie Team)$$

Figure 3.2: Queries for Section 3.6.7

We created four datasets (denoted by $SD1$ to $SD4$) for the scientific database as follows. $SD4$ is the original scientific database, and each of the remaining datasets are subsets of SD varying in size created such that they satisfy the following two properties: (1) for $i \in [1, 3]$, we have $|SD_i| = \frac{i}{4} \times |SD_4|$ and (2) for each query Q on the scientific database, $Q(SD1) \subseteq Q(SD2) \subseteq Q(SD3) \subseteq Q(SD)$. Similarly, we created three datasets (denoted by $BB1$ to $BB3$) for the baseball database with the similar properties.

The properties of the datasets and query results are shown in Table 3.9. For convenience, the two relations in the scientific database are abbreviated as P and S, and the three relations in the baseball database are abbreviated as T, B, and M.

Scientific Data	SD1	SD2	SD3	SD4
Size(P)	1000	2000	3000	3926
Size(S)	111	221	316	424
# of $Q_1(D)$	1	1	1	1
# of $Q_2(D)$	3	6	7	8
# of $Q_3(D)$	1	2	4	6
# of $Q_4(D)$	2	3	4	4
# of $Q_5(D)$	3	4	6	6

(a) Statistics for Scientific Database

Baseball Data	BB1	BB2	BB3
Size(T)	10	20	30
Size(B)	350	751	1034
Size(M)	10	24	33
# of $Q_6(D)$	5	7	9
# of $Q_7(D)$	2	3	4
# of $Q_8(D)$	2	2	2
# of $Q_9(D)$	2	3	4

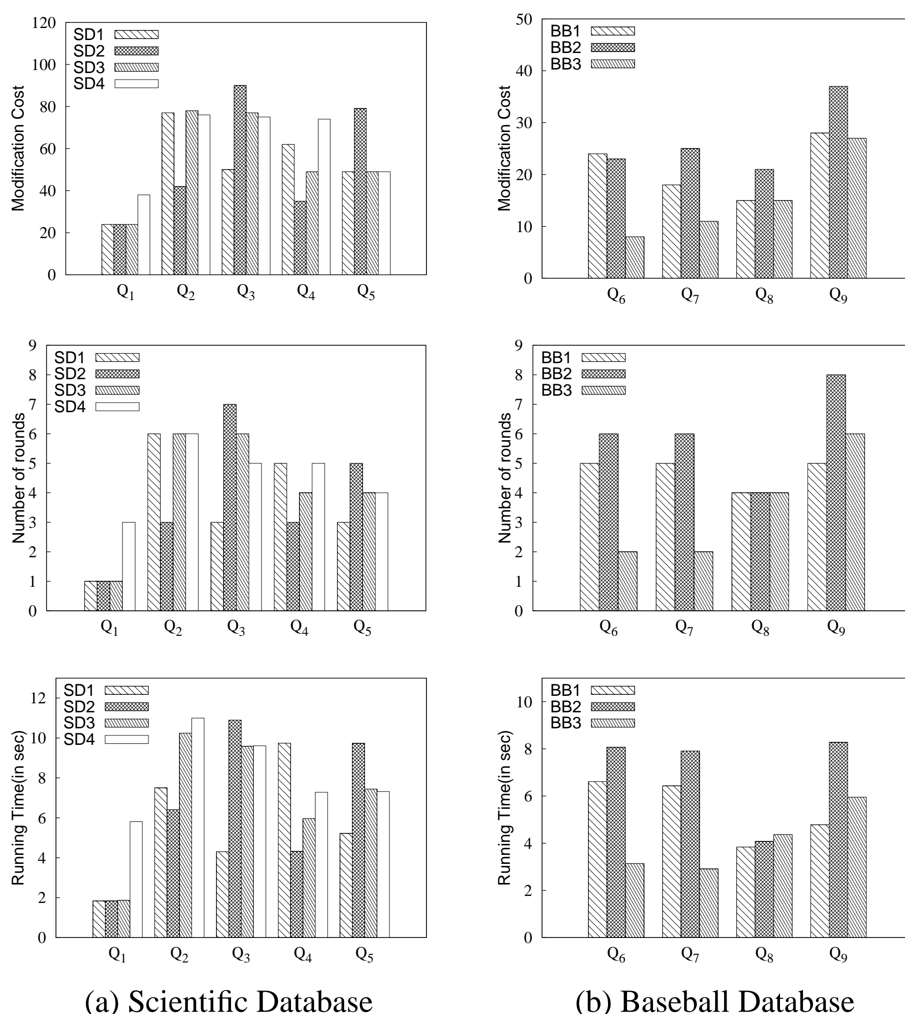
(b) Statistics for Baseball Database

Table 3.9: Properties of datasets and query results

The experimental results are shown in Figure 3.3. For each dataset, we show the total modification cost, the number of iterations to find the intended query, and the execution time. As shown in the Figure 3.3, the effect of the initial database-result pair on performance does not not have a clear trend. For example, Q_6 and Q_7 on $BB3$ incurred the lowest modification cost and number of iterations. However, Q_8 and Q_9 on $BB1$ incurred the lowest modification cost and number of iterations. In terms of the number of iterations, $BB2$ required the largest number, but in terms of the modification cost, Q_6 on $BB2$ outperforms Q_6 on $BB1$. As for the scientific dataset, $SD2$ incurred the lowest modification cost and number of iterations for Q_2 and Q_4. However, for Q_3 and Q_5, their performance on $SD2$ is the worst. In summary, there is no clear trend for the effect of the initial database-result pair on the performance.

3.6.8 Effect of Size & Entropy of Attributes' Active Domains

In this section, we present additional experimental results to evaluate the effect of the size and entropy of the active domains of attributes on performance.



(a) Scientific Database (b) Baseball Database

Figure 3.3: Effect of initial database-result pair

For this experiment on the scientific database, we created 5 datasets (denoted by $D_1, \cdots, D_5$) by varying the number of distinct values of a chosen attribute (denoted by A). The total number of distinct values for A for these datasets are shown in Table 3.10. D_1 is the

Data sets	D_1	D_2	D_3	D_4	D_5
# of distinct values	3725	2978	2230	1490	749
% reduced	0%	20%	40%	60%	80%

Table 3.10: Number of distinct values for attribute A in datasets

original dataset, and the each of the other datasets was created by reducing the number of distinct attribute values for A by a certain percentage as shown in Table 3.10. This is done by replacing each eliminated attribute value by one of the existing attribute values

such that the following properties hold: (1) let T_i denote the instance of relation T in D_i, we have $\pi_A(T_i) \supset \pi_A(T_{i+1})$, $i \in [1, 5)$. (2) for each intended query Q on the scientific database, $Q(D_i) = Q(D_j)$ for any $i, j \in [1, 5]$.

For each of two intended queries, Q_1 and Q_2, on the scientific database, and for each dataset D_j, $j \in [1, 5]$, we used the Query Generator module to generate a set of candidate queries (denoted by $QC(Q_i, D_j)$) for the input database-result pair $(D_j, Q_i(D_j))$. There were a total of 19 common candidate queries for Q_1 and a total of 18 common candidate queries for Q_2; i.e., $|S_1| = 19$ and $|S_2| = 18$, where $S_i = \bigcap_{j=1}^{5} QC(Q_i, D_j)$.

Table 3.11 shows the performance of each of the 19 candidate queries in S_1 as intended query on each of the 5 datasets. As before, the performance is measured in terms of the number of iterations and total modification cost to identify the intended query. Similarly, Table 3.12 shows the performance results for the candidate queries in S_2.

Observe that for the same query, the performance results on the datasets D_2 to D_5 are mostly the same. For the datasets D_1 and D_2, we observe that some queries perform better on D_1 while other queries perform better on D_2. In summary, our experimental results show that there is no clear trend for the effect of the size and entropy of the attributes' active domains on performance.

3.6.9 User Study

In this section, we present the results of a user study conducted with 3 participants (all of whom were CS graduate students) to evaluate the feasibility of our approach. The screen capture of the system UI is shown in Figure 3.4. The interface first showed the input database-result pair to the user. The user can scroll up and down to browse the tuples in database and query result. In each iteration, the system highlighted the differences between original and modified tuples. We used different colors to mark the modified

Query ID	1	2	3	4	5	6	7	8	9	10
D_1	6	6	4	6	6	3	6	4	6	6
D_2	3	8	4	8	7	3	7	5	7	6
D_3	3	8	4	8	7	3	7	5	7	6
D_4	3	8	4	8	7	3	7	5	7	6
D_5	3	8	4	8	7	3	7	5	7	6

Query ID	11	12	13	14	15	16	17	18	19
D_1	3	3	6	4	6	3	6	3	3
D_2	3	4	5	5	7	4	3	3	3
D_3	3	4	5	5	7	4	3	3	3
D_4	3	4	5	5	7	4	3	3	3
D_5	3	4	5	5	7	4	3	3	3

(a) Number of iterations

Query ID	1	2	3	4	5	6	7	8	9	10
D_1	155	155	52	155	155	38	155	52	79	155
D_2	38	113	64	113	97	38	97	79	97	85
D_3	38	113	64	113	97	38	97	79	97	85
D_4	38	113	64	113	97	38	97	79	97	85
D_5	36	113	64	113	97	38	97	79	97	85

Query ID	11	12	13	14	15	16	17	18	19
D_1	38	38	155	52	155	38	79	38	38
D_2	38	64	79	79	97	64	38	38	38
D_3	38	64	79	79	97	64	38	38	38
D_4	38	64	79	79	97	64	38	38	38
D_5	38	64	79	79	97	64	36	38	38

(b) Modification Cost

Table 3.11: Effect of size & entropy of active attribute domain for query Q_1

Query ID	1	2	3	4	5	6	7	8	9
D_1	4	5	5	6	6	5	3	3	3
D_2	4	6	5	6	6	5	4	4	4
D_3	4	5	4	4	5	5	4	4	4
D_4	4	5	4	4	5	5	4	4	4
D_5	4	5	4	4	5	5	4	4	4

Query ID	10	11	12	13	14	15	16	17	18
D_1	2	2	2	3	3	4	3	4	3
D_2	4	3	4	4	3	3	3	3	4
D_3	3	4	4	4	3	3	3	3	4
D_4	3	4	4	4	3	3	3	3	4
D_5	3	4	4	4	3	3	3	3	4

(a) Number of iterations

Query ID	1	2	3	4	5	6	7	8	9
D_1	47	60	60	74	74	60	79	79	79
D_2	63	81	63	82	82	63	61	61	59
D_3	63	82	66	66	82	80	60	60	60
D_4	63	82	66	66	82	80	60	60	60
D_5	52	82	66	66	82	80	60	60	60

Query ID	10	11	12	13	14	15	16	17	18
D_1	55	55	55	67	46	58	46	58	67
D_2	59	49	49	49	49	49	38	49	49
D_3	48	60	49	49	49	49	53	49	49
D_4	48	60	49	49	49	49	53	49	49
D_5	48	60	49	49	49	49	53	49	49

(b) Modification Cost

Table 3.12: Effect of size & entropy of active attribute domain for query Q_2

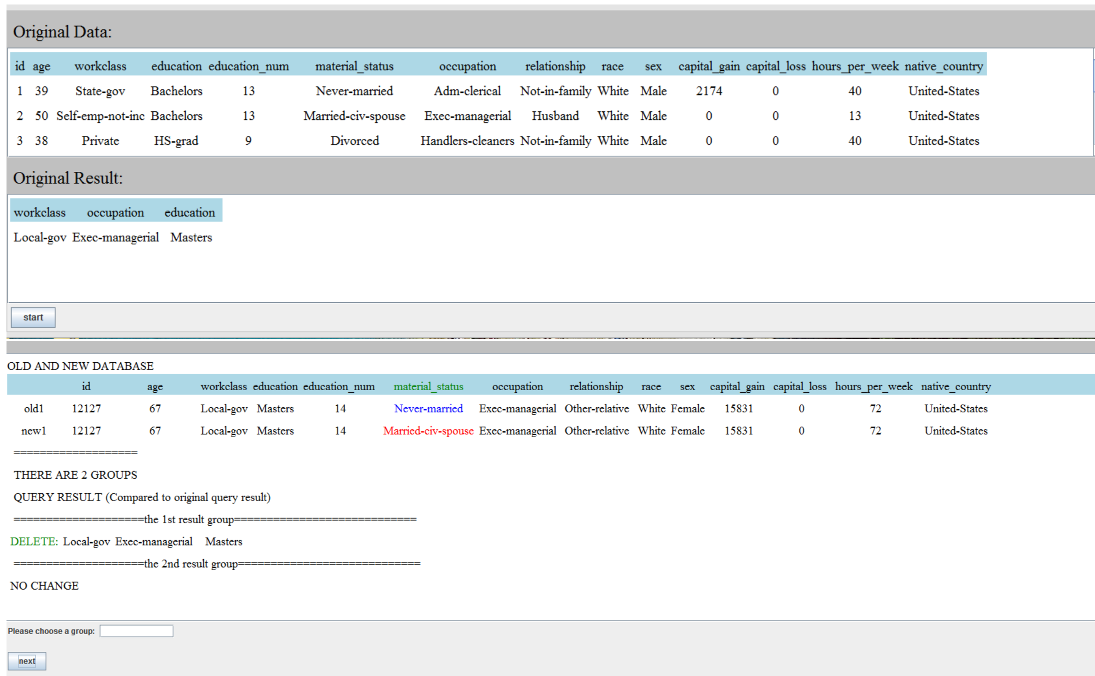


Figure 3.4: UI screen capture

attribute, the old and updated values to help users examine the modifications. We also showed the user the groups of query results and asked the user to choose the group with correct result. Once the user selected a group, we used the corresponding queries as candidates for the next generation.

For this experiment, we used the the Adult relation (containing 5227 tuples) extracted from the 1994 Census database[5]. This dataset was chosen over the scientific and baseball datasets as we felt that its data domain would be easier to understand for users. The following three queries were used for this experiment.

- Q1 Find *workclass*, *occupation* and *education* for white females who are at least 64 years old, never married, and with a capital gain of more than 500.
- Q2 Find *education*, *occupation* and *hours-per-week* for people whose native country is Taiwan and occupation is Armed-Forces.

[5]http://archive.ics.uci.edu/ml/datasets/Adult

Q3 Find age and $education$ for people whose native country is England and occupation is Tech-support.

The query result sizes for $Q1$ to $Q3$ are, respectively, 1, 2 and 1 tuple.

We compare Q-QFE against an alternative strategy Q-QFE' in terms of number of iterations, modification cost and user's response time. Instead of guided by current cost model to find the cheapest modifications, in each iteration, Q-QFE' picks the modifications that can split queries into the most subsets. Intuitively, this strategy could decrease the size of query subset, and reduce the number of iterations.

Table 3.13 shows the per-round statistics for each query using different strategies: "#Queries" refers to the number of candidate queries at the start of an iteration, "#subsets" refers to the number of query subsets after an iteration and "Cost" refers to the total modification cost for an iteration. Observe that Q1 incurs the highest modification cost of 27 using Q-QFE' while Q3 has the lowest modification cost of 12 using Q-QFE. For one iteration, maximum modification cost of Q-QFE is 5, while the maximum cost of Q-QFE' is 22. Although Q-QFE' reduces the number of iterations to 2 for each query, the total modification cost is still higher than Q-QFE.

Query		**Q-QFE**					**Q-QFE'**	
	i-th iteration	1	2	3	4	5	1	2
Q1	#Queries	10	5	3	2	-	10	5
	#subsets	2	2	2	2	-	2	5
	cost	5	4	4	4	-	5	22
Q2	CQ Size	17	10	5	3	2	17	2
	# of subsets	2	2	2	2	-	5	2
	cost	4	4	5	4	4	22	4
Q3	CQ Size	11	5	3	2	-	11	5
	# of subsets	2	2	2	2	-	2	5
	cost	3	3	3	3	-	3	17

Table 3.13: Per-round statistics for queries

Table 3.14 shows the experimental results for two of the queries, Q1 and Q3, which, respectively, took the longest and shortest times among the 4 queries. The time taken

User		Q-QFE				Q-QFE'	
	i-th iteration	1	2	3	4	1	2
1	Utime	17.7	9.1	4.7	28.4	79.2	84.3
	Stime	1.3	1.2	0.4	0.1	1.2	1.2
	Ttime	19.0	10.3	5.2	28.5	80.5	85.5
2	Utime	50.4	15.8	26.8	29.7	26.5	85.8
	Stime	1.3	1.2	0.4	0.1	1.2	1.2
	Ttime	51.7	18.1	27.2	29.8	27.7	87.0
3	Utime	19.3	8.1	6.1	25.2	31.9	75.7
	Stime	1.3	1.2	0.4	0.1	1.2	1.2
	Ttime	20.6	9.3	6.5	25.3	33.1	76.9

(a) Time to find Q1

User		Q-QFE				Q-QFE'	
	i-th iteration	1	2	3	4	1	2
1	Utime	11.2	2.7	1.8	2.1	79.2	84.3
	Stime	1.2	1.2	0.5	0.1	1.2	1.3
	Ttime	12.4	3.9	2.3	2.2	80.4	85.6
2	Utime	23.7	8.7	9.2	8.7	17.2	40.3
	Stime	1.2	1.2	0.5	0.1	1.2	1.3
	Ttime	24.9	9.9	9.7	8.8	18.4	41.6
3	Utime	9.9	4.6	3.6	3.7	18.1	22.9
	Stime	1.2	1.2	0.5	0.1	1.2	1.3
	Ttime	11.2	5.8	4.1	3.8	19.3	24.2

(b) Time to find Q3

Utime: user response time; Stime: system running time; Ttime: total time

Table 3.14: Timing results for user study (in secs)

by the Query Generator module (around 0.5 seconds) is not included in the timings for the first iteration. Observe that the user response time dominates the total time taken for each iteration. The longest and shortest user response times are, respectively, around 85 seconds and 2 seconds. Overall, the user study experiment demonstrates that the users were able to effectively determine the intended queries with reasonable effort.

Comparing Q-QFE and Q-QFE', it is obvious that users are quite sensitive to the modification cost. The response time using Q-QFE' is much higher than using Q-QFE. Even the number of iterations is less, the total time is still much higher. E.g., it takes the first user 63 seconds to find Q1 using Q-QFE, while using Q-QFE' it takes 166 seconds. Therefore,

our cost model is very practical in terms of user's response time.

3.7 Conclusion

In this chapter, we have developed a new approach, called Query from Examples (QFE), to help non-expert database users construct SQL queries. We also propose a Query-based approach of QFE (Q-QFE). Our approach does not expect users to be familiar with SQL and only requires that users are able to determine whether a given output table is the result of his or her intended query on a given input database. Using an initial user-specified pair of database D and output table for the user's target query on D, Q-QFE is able to identify the user's target query through a sequence of rounds of interactions with the user. Each interaction round obtains feedback from the user to identify the correct output result for a modified database that is judiciously generated to minimize the user's effort to provide feedback.

Our experimental evaluation of Q-QFE demonstrates the feasibility of our approach and the effectiveness of our techniques. As part of our future work, we plan to extend our approach to support more expressive queries and explore optimization techniques to improve performance. In addition, we also plan to conduct a more extensive user study to evaluate the approach's effectiveness.

CHAPTER 4

SCHEMA-BASED APPROACH

In the previous chapter, we described a novel Query-based approach for QFE, to help non-expert database users who are not sophisticated with SQL construct queries. It takes a database-result pair as input, and generates a set of candidate queries with the *Query Generator module* at first. Then the *Database Generator module* distinguishes those queries iteratively. Finally, the approach outputs the user's target query.

In this chapter, we describe a schema-based approach of QFE (S-QFE) to generate candidate queries from a given database-result pair (D, R). Different from Query-based approach, we adopt an iterative method to identify the target query schema first. We first introduce the problem in Section 4.1, followed by the approach overview in Section 4.2. The details are discussed in Section 4.3 to Section 4.5. An experimental evaluation is presented in Section 4.6. Finally, we conclude in Section 4.7. The notations we use throughout this Chapter is shown in Table 4.1.

Notation	Description
Q	Query
D	Database
D'	Modified database
R	Query result
$Q(D)$	Query Q's result on database D
A	Attribute
JS	Join schema
JR	Set of join relations
PA	Set of projection attributes
SA	Set of selection attributes
JP	Set of join predicates
SP	Set of selection predicate
QS	Query schema
qs-query	Query with the query schema qs
$\mathcal{S}$	Set of candidate query schemas
$J(D)/J_{qs}(D)$	Result of joining all the join relations in qs
J_{qs}^{+}	Positive partitions of $J(D)$
J_{qs}^{-}	Negative partitions of $J(D)$
J_{qs}^{free}	Free partitions of $J(D)$
$domain(A)$	Domain of attribute A
QS_{min}^{qs}	Minimal query of query schema qs
$iscore(A)$	Impact score of attribute A

Table 4.1: Notation table of Chapter 4

4.1 Introduction

Recently, a number of works [64, 70, 61] have been proposed to handle the query reverse engine problem which focuses on deriving the query Q such that $Q(D) = R$, where D and R are from user's input. In the previous chapter, we proposed a Query-based approach of QFE (Q-QFE), which generates all the candidate queries first, and then we help the user to get the intended one. Recall that we used QBO [64] as the the *Query Generator module* in Chapter 3. One drawback of using QBO is that it might generate too many queries which increase the burden on the *Database Generator module*. Here is an example.

Example 4.1. *Consider the baseball dataset, which contains 9 relations. The number of attributes in each relation vary from 3 to 29. Among them, relation "Manager" has 11 attributes and "Team" has 29 attributes. To find the following target query, QBO generated*

more than 90 queries in total. We show three of the generated queries in Figure 4.1.

```
SELECT distinct Manager.managerID, Team.year, Team.rank
FROM Manager, Team
WHERE Manager.teamID = Team.teamID AND Manager.year = Team.year
AND Team.teamID = 'CIN' AND Team.year > 1982 AND Team.year < 1988;
```

$$\begin{array}{l}
Q_1 : SELECT\ distinct Manager.managerID, Team.year, Team.rank \\
\quad FROM\ Manager, Team \\
\quad WHERE\ Team.teamID = Manager.teamID\ AND\ Team.year = Manager.year AND \\
\qquad ((Team.franchID = “CIN''\ AND\ Team.BB \leq 563\ AND\ Team.HR > 191\ AND\ Team.E > 113)\ OR \\
\qquad (Team.franchID = “CIN''\ AND\ Team.BB > 563\ AND\ Team.R \leq 677)\ OR \\
\qquad (Team.franchID = “CIN''\ AND\ Team.BB > 563\ AND\ Team.R > 677\ AND\ Manager.plyrMgr \neq “N''));
\end{array}$$

$$\begin{array}{l}
Q_2 : SELECT\ distinct Manager.managerID, Team.year, Team.rank \\
\quad FROM\ Batting, Team, Master, Manager \\
\quad WHERE\ Master.playerID = Batting.playerID\ AND\ Team.teamID = Batting.teamID\ AND \\
\qquad Team.year = Batting.year\ AND\ Master.playerID = Manager.playerID\ AND \\
\qquad ((Manager.plyrMgr \neq “N''\ AND\ Manager.G \leq 161\ AND\ Team.BBA > 577\ AND \\
\qquad Team.BBA \leq 578\ AND\ Manager.lgID \neq “”L'')\ OR \\
\qquad (Manager.plyrMgr \neq “N''\ AND\ Manager.G > 161\ AND\ Batting.2B \leq 12\ AND \\
\qquad Team.HR > 106\ AND\ Manager.plyrMgr = “Y''));
\end{array}$$

$$\begin{array}{l}
Q_3 : SELECT\ distinct Manager.managerID, Team.year, Team.rank \\
\quad FROM\ Fielding, Team, Master, Manager \\
\quad WHERE\ Master.playerID = Fielding.playerID\ AND\ Team.teamID = Fielding.teamID\ AND \\
\qquad Team.year = Fielding.year\ AND\ Master.playerID = Manager.playerID\ AND \\
\qquad ((Team.SO > 855\ AND\ Manager.plyrMgr \neq “N''\ AND\ Team.SO \leq 856\ AND\ Manager.W > 86)\ OR \\
\qquad (Team.SO > 855\ AND\ Manager.plyrMgr \neq “N''”\ AND\ Team.SO > 856\ AND\ Team.HA > 1443\ AND \\
\qquad Team.HA \leq 1465\ AND\ Manager.W \leq 86\ AND\ Manager.plyrMgr = “Y''));
\end{array}$$

Figure 4.1: Queries generated by QBO

As shown above, these queries are quite different from the target query, although they can get the same query results. In the queries generated by QBO, there are 7 different join schemas involving 2, 3 or 4 relations, and for each join schema there are more than 10 queries generated. Besides overburdening the Database Generator module, to generate so many candidate queries is also very time consuming.

To avoid generating too many candidate queries, in this chapter, we propose a *Schema-based approach* of QFE (S-QFE) to help non-expert users construct the target query. In

the beginning, S-QFE asks the user to provide an initial database-result pair (D, R) of the target query Q_* (i.e., $Q_*(D) = R$). As there will be many queries under different query schemas that can transform D to R, S-QFE first computes a set of candidate query schemas, and then asks the user to identify the target query schema by changing the database D and showing new database-result pairs iteratively. At each iteration, we present a modified database D', and the user examines new database-result pairs to determine if it is correct with respect to his or her intended query. By getting the user's feedback on a series of database-result pairs, our approach can identify the target query schema, and we continue to generate candidate queries with the target query schema.

4.2 Approach Overview

For the ease of presentation, we first give the definition of query schema, and then we introduce our approach overview. For simplicity, we only consider SQL queries without aggregate function at first.

Definition 4.1. *Consider a SPJ SQL query Q, which can be expressed as a 5-tuple, $(JR$, JP, PA, SA, $SP)$, where JR is a set of joined relations in Q; JP is a set of join predicates for JR; PA is a list of projection attributes; SA is a set of selection predicate attributes and SP is a set of selection predicates in Q. We refer to (JR, JP, PA, SA), without SP, as the* ***query schema*** *of Q. We refer to Q as a qs-query if its query schema is qs.*

To simplify the discussion, we mainly focus on identifying SA of the target query schema, which is the most complex problem. We assume that the PA and JR are the same from all the candidate query schemas, and all the relations in JR are joined based on foreign-key relationships. We will discuss how to relax these assumptions in Section 4.5.

Given a database-result pair (D, R), a query schema qs is defined to be *valid* if there exists at least one qs-query Q whose query results $Q(D) = R$. Otherwise, qs is an *invalid* query schema. Given a pair of (D, R), there may exist different valid query schemas, and there could be multiple queries sharing the same query schema that can generate the same query result as R. To avoid generating all those queries, we propose S-QFE to identify the correct query schema first. The overview of our approach is shown in Figure 4.2.

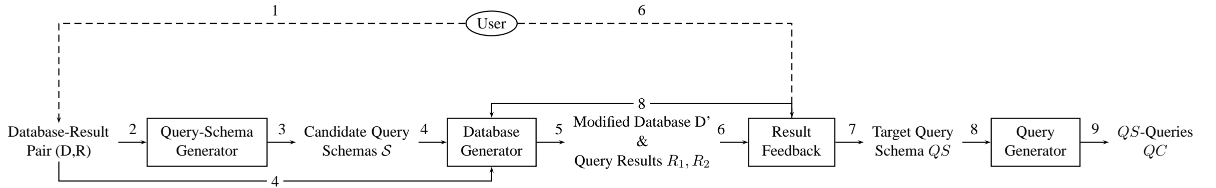


Figure 4.2: Overall Architecture of schema-based QFE

S-QFE first obtains an initial database-result pair (D, R) from the user where R is the result of the users target query on the database D. The *Query-Schema Generator module* takes (D, R) as input to generate a set of valid query schemas $\mathcal{S} = \{qs_1, \cdots, qs_n\}$ for (D, R); i.e., $\forall qs_i \in \mathcal{S}, \exists qs_i$-query $Q_i : Q_i(D) = R$. To efficiently identify the user's target query schema, QFE iteratively modifies the database and presents new database-result pairs to the user. At each iteration, the *Database Generator module* modifies a tuple t in database D to t', where t is a tuple satisfies all the query's selection predicates. For the ease of description, we say a tuple t is *in the query result*, if t satisfies all the query's selection predicates. The modified t' should partition the query schemas $\mathcal{S}$ into two groups $\mathcal{S}_1$ and $\mathcal{S}_2$ as follows: (1) t' is in the query result for all the query schemas in $\mathcal{S}_1$; (2) t' is not in the result for any query schemas in group $\mathcal{S}_2$. Next, we ask the user whether the correct query result should contain t' with respect to the modified database. If the user's feedback is yes, we eliminate the query schemas in group $\mathcal{S}_2$. Otherwise, we eliminate the schemas in group $\mathcal{S}_1$. We continue this process until we could identify the target query schema. Once we obtain the target schema qs, we can generate the qs-queries

whose query result is R as the candidate queries. To find the target query, we could simply use Q-QFE as described in Chapter 3.

As discussed before, we identify the query schema by modifying tuples to partition the candidate query schemas into different groups in terms of their query results. There are two main challenges. The first challenge is how to find the candidate valid query schemas. Given a database D with n attributes, the number of possible selection-attribute set is $2^n - 1$, which could be very large. Hence, it is not practical to consider all of them as the candidates. The second one is how to choose the tuple t and what new values should be set. To solve the problem, we propose a two-step approach to identify the query schema. At the first step, we compute the candidate query schemas by eliminating invalid query schemas, and at the second step we choose a tuple and calculate the new values which can group query schema into different groups. Based on the user's feedback, we continue modifying database until we identify the target query schema. The approach is shown in Algorithm 4.1

Algorithm 4.1: QFE: Schema-based approach

Input: A database-result pair (D, R)
Output: Candidate queries QC

1. Let G be the join graph of all relations in the database D
2. PA = Map-Projection-Attributes(D, R)
3. **foreach** $pa \in PA$ **do**
4.   Let $rels$ be the relations where attributes pa are from
5.   **foreach** subgraph $JS$of G which is a valid join schema **do**
6.   build joined relation $J(D)$ with all the relations in JS
7. $\mathcal{S}$ = Query-Schema-Generator$(J(D), R)$
8.   **while** $|\mathcal{S}| > 1$ **do**
9. D' = Database-Generator($D, \mathcal{S}$)
10. D' partitions the schemas into groups $\mathcal{S}_1, \mathcal{S}_2$ with different query results
11. x= Result-Feedback(D') // $x \in \{1, 2\}$
12. $\mathcal{S} = \mathcal{S}_x$
13.   **if** $|\mathcal{S}| \neq 0$ **then**
14. QS be the query schema in $\mathcal{S}$
15.   break
16. QC = Query-Generator(D, R, QS)
17. **return** QC

Algorithm 4.1 takes a database-result pair (D, R) as input, and returns the candidate queries QC as output. Given the query result, we first compute the join graph G of database D according to the foreign-key relationships (line 1). Then we find all the projection attributes PA, where PA is a set of attribute set $\{(A_1, A_2, \cdots, A_n)\}$, the i-th column of R is a projection of attribute A_i in D, and $A_i \neq A_j (i \neq j)$. The function `Map-Projection-Attributes` uses a brute-force method to compute PA (line 2). For simplicity, we omit the details of function `Map-Projection-Attributes` here. For each projection-attribute set pa, we find the relations $rels$ containing all attributes in pa (line 3). It is clear that the target query schema must contains all the relations of $rels$. Consider a subgraph JS of G, if it involves all the relations of $rels$, we say JS is a valid join schema. For each valid join schema, we compute the joined relation $J(D)$ by joining all the relations in JS, and `Query-Schema-Generator` computes the candidate query schemas (lines 5 to 7).

S-QFE winnows the candidates iteratively using a divide-and-conquer strategy. At each iteration, the `Database-Generator` takes as inputs (D, R) and candidate query schemas $\mathcal{S}$, which is the set of remaining candidates at the start of the iteration, to generate a new database D'. D' will partition candidates into two groups, and ask the user to select the correct result (lines 8 to 11). According to the user's feedback, S-QFE will start another iteration using the subset of candidates $\mathcal{S}_x$ corresponding to x if $\mathcal{S}_x$ has more than 1 query schemas (line 12). Otherwise, S-QFE terminates with the only query schema as the target query schema (line 14). In the event that none of the query schema is correct, it means the $\mathcal{S}$ will be an empty set in the end. In this case, we will start another round with new valid join schema or new projection-attribute set.

Once the target query schema is identified, we continue to generate the candidate queries QC (line 16). Because so far we only consider the SPJ query, given a target query schema, there will only be one query Q could get the result R on database D.

Before delving into the details, we first introduce several notions, which we use through-

out the whole chapter. Here we borrow the notation of *data partition* from QBO [64].

Given a query schema $qs = (JR, JP, PA, SA)$ and a database-result pair (D, R), let $J_{qs}(D)$ denote the result of joining all the relations in JR using join predicates JP with respect to D. Suppose that there are k distinct tuples in R with $R = \{r_1, \cdots, r_k\}$. Then R can partition the tuples in $J_{qs}(D)$ into $k+1$ partitions, $P_0, \cdots, P_k$, where P_0 could be empty, with the following properties: (1) for each P_i, $i \neq 0$, each tuple in P_i can generate the output tuple $r_i \in R$; and (2) each tuple in P_0 does not generate any output tuple in R.

Note that, if the target query is under bag semantics, it is possible that there are duplicated records in R. To handle the bag-semantics query, we only use the distinct value in R to partition the tuples in $J_{qs}(D)$. If R contains duplicate tuples, e.g., r_i equals to r_j, then they correspond to the same partition. In this way, we partition tuples into multiple partitions without overlap under either bag or set semantics.

We can classify the partitions in $J_{qs}(D)$ into three types: P_0 is a *n*egative partition; P_i is a *p*ositive partition if $i > 0$ and $|P_i| = 1$; otherwise, P_i is a *f*ree partition. A tuple $t \in J_{qs}(D)$ is classified as a negative/positive/free tuple if t is in a negative/positive/free partition. Let $J_{qs}(D) = J^-_{qs} \cup J^+_{qs} \cup J^{free}_{qs}$, where J^-_{qs}, J^+_{qs}, and J^{free}_{qs}, respectively, denote the subset of negative, positive, and free tuples in $J_{qs}(D)$.

The intuition of our approach is based on the following observation. Consider two query schemas qs and qs', where qs contains attribute A and qs' does not. Let t be a positive tuple that can generate output record r in query result R. Now let us modify t's attribute value of A from v to v' such that t would not be selected by any candidate query sharing query schema qs, and then ask the user whether r should appear in the query result with respect to the modified data. If the target query schema is qs, then r would not be in the query result of the modified database; if the target query schema is qs', then r should still be in the query result. Thus, we can identify whether A is an attribute in target query schema, and we refer to v' as an *invalid*-(qs, A) *value*.

In the following sections, we will discuss our approach for two cases. The first one is there exists at least one positive partition with respect to given (D, R), and the second case is only free and negative partitions exist in database D. The reason is that each free partition contains multiple records having same projection values. Without positive partition, there could be many different combination of tuples to generate the query result R, and we can not find a positive tuple to modify, which increase the complexity.

4.2.1 Limitation

Note that it is possible that we can not derive the target query schema based on the given database-result pair because of the constraint of the given database-result pair. Here is an example.

Example 4.2. *Suppose that the user needs help to compose the query for the following database-result pair (D, R), where D consists of a single table. The user's target query is "find the male employee in IT department whose salary is more than 4500".*

Employee

Eid	*name*	*gender*	*dept*	*salary*
1	*Alice*	*F*	*Sales*	*4700*
2	*Bob*	*M*	*IT*	*4700*
3	*Caleb*	*M*	*Service*	*5000*
4	*Darren*	*M*	*IT*	*5000*
5	*Elly*	*F*	*IT*	*4700*

Database D

name
Bob
Darren

Result R

As mentioned in the example, the user's target query contains three predicates "$dept = $ 'IT'", "$gender =$ 'M'" and "$salary > 4500$". With the given database D, we can not find a record that a male employee works in IT department whose salary is less than 4500.

Therefore, when we compose the query schema, we can easily construct a valid query with predicates "$dept = `IT'$" and "$gender = `M'$" without constraints on attribute "$salary$". It is difficult to deduce a predicate with attribute "$salary$" due to the absence of counter example.

As the example shows, to find the target query, for each selection predicate in the target query, the given database should contain at least one negative tuple which violates the predicate. Otherwise, we can not derive the predicate from the given database. For simplicity, we assume the given database is sufficient to derive the target query for the rest of the chapter.

4.3 Handling The Scenario With Positive Partition

In this section, we discuss how to find the target query schema if there exists at least one positive partition in the database. We present our approach as the procedures shown in Algorithm 4.1. We first discuss how to prune the invalid query schema, and then introduce an approach to modify the database to identify the target query schema.

4.3.1 Algorithm Query-Schema-Generator

In this section, we discuss how to compute the candidate query schemas. To facilitate the explanation of our approach, we first introduce the notion of *minimal query schema*.

Definition 4.2. ***(minimal query schema)*** *Given a database-result pair (D, R), a query schema $qs = (JR, JP, PA, SA)$ is defined to be a minimal query schema if*

1. *qs is a valid query schema;*

2. *for every non-empty proper subset SA' of SA, query schema (JR, JP, PA, SA') is not a valid query schema.*

Lemma 4.1. *If $qs = (JR, JP, PA, SA)$ is a minimal query schema, then for any $SA' \supset SA$, $qs' = (JR, JP, PA, SA')$ is also a valid query schema.*

Proof. Suppose qs is a minimal query schema with selection attributes SA, and qs' is a query schema whose $SA' = SA \cup \{A_i\}$, where $A_i \notin SA$. There exists at least one valid qs-query Q. We can construct a qs'-query Q' by adding new selection predication $range(A_i) = domain(A_i)$, where $domain(A_i)$ is the domain of attribute A_i. Thus, Q' is also a valid query and qs' is a valid query schema. □

According to Lemma 4.1, to find all the valid query schemas, one reasonable method is to identify all the minimal query schemas first, then we can easily append selection attributes to get more valid query schemas. In this work, we adopt an elimination method to get the candidate query schemas. We first introduce *minimal query*, which can be used to test the validity of a given query schema. Then, we present our approach to compute the candidate query schemas.

Definition 4.3. ***(minimal query)*** *Given a database-result pair (D, R) and a query schema qs, let $FP = \{FP_1, \cdots, FP_m\}$ denote the collection of free partitions with respect to qs and D. We define Q^{qs}_{min} to be the minimal query belonging to qs (or minimal qs-query for short) if the set of selection predicates in Q^{qs}_{min} is given by $\{A_i \in [\ell_i, u_i] : A_i \in SA\}$, where $\ell_i = \min\{\min\{\pi_{A_i}(J^+_{qs})\}, \max\{\pi_{A_i}(FP_1)\}, \cdots, \max\{\pi_{A_i}(FP_m)\}\}$ and $u_i = \max\{\max\{\pi_{A_i}(J^+_{qs})\}, \min\{\pi_{A_i}(FP_1)\}, \cdots, \min\{\pi_{A_i}(FP_m)\}\}$.*

Lemma 4.2. *Given a database-result pair (D, R), if the target query schema is qs, then $Q^{qs}_{min}(D) \subseteq R$.*

Proof. According to the definition of minimal query, given a query schema qs, for each selection attribute, the selection predicate in Q^{qs}_{min} is derived by only the positive partitions.

Thus, if qs is the target query schema, the target query Q's selection predicates must be looser than Q^{qs}_{min}. Otherwise, positive tuples will not satisfy the conditions. Hence, $Q^{qs}_{min}(D) \subseteq R$. □

With Lemma 4.2, we can prune invalid query schema as follows: given a database-result pair (D, R), if $Q^{qs}_{min}(D) - R \neq \emptyset$, then qs cannot be the target query schema, otherwise qs is referred to as a *candidate query schema*. We compute all the candidate query schemas and the algorithm is shown in Algorithm 4.2.

Algorithm 4.2: Query-Schema-Generator

```
   Input: join relation J(D), query result R
   Output: Candidate query schemas S
1  Initialize S = ∅, QS_1 = ∅
2  Let AS be the set of all the attributes in the joined relation J(D)
3  foreach attribute A ∈ AS do
4      Construct query schema qs whose SA = {A}
5      if Q^{qs}_{min}(J(D)) − R ≠ ∅ then
6          QS_1 = QS_1 ∪ {qs}
7      else
8          S = S ∪ {qs}
9  QS_2 = QS_1
10 while QS_2 ≠ ∅ do
11     Let QS_3 = ∅
12     foreach qs_2 ∈ QS_2 do
13         foreach qs_1 ∈ QS_1 do
14             Let atts be the superset of qs_1 and qs_2's selection attributes
15             if atts has been computed before then
16                 continue
17             Construct query schema qs whose SA = atts
18             if Q^{qs}_{min}(J(D)) − R ≠ ∅ then
19                 QS_3 = QS_3 ∪ {qs}
20             else
21                 S = S ∪ {qs}
22     QS_2 = QS_3
23 return S
```

As shown in Algorithm 4.2, we adopt a bottom-up approach to compute the candidate query schema. Taking join relation $J(D)$ and query output R as input, Algorithm 4.2 uses all the attributes in $J(D)$ to compute the candidate query schemas (line 2). We enumerate the query schema by gradually increasing the number of selection attributes. First we

examine the query schemas with only one selection attribute (lines 3 to 8). For each query schema, we construct the minimal query and test whether it is a candidate query schema (line 5). If the answer is yes, we add it into $\mathcal{S}$, and stop appending more attribute to this query schema. Otherwise, we append more selection attributes to enumerate more query schemas (lines 10 to 22). We add one more attribute each time to the selection attributes of the invalid query schema (line 14). If the selection-attributes set has been computed before, we do not need to construct minimal query to examine it again (lines 15 to 16). Then we examine the query schema with minimal query and store all the candidate query schemas as before (lines 18 to 24). The whole algorithm terminates when we find all the potential minimal query schema, and the time complexity is $O(2^n)$, where n is the number of selection attributes SA.

4.3.2 Algorithm Database-Generator

In this section, we present the details of the function `Database-Generator`. Recall that we define *invalid-*(qs, A) *value* in Section 4.2: given a positive tuple t and a query schema qs containing selection attribute A, if we change t's A value to a value v', such that t becomes a negative tuple, we refer to v' as an *invalid-*(qs, A) *value*. Conversely, if value v' keeps t as a positive tuple, v' is called a *valid-*(qs, A) *value*.

Consider two valid query schemas qs and qs', where attribute A is in selection attributes of qs and not in qs''s. To distinguish these two query schemas, we need to determine an *invalid-*(qs, A) *value* for attribute A. Then we can modify the database with the invalid value, and ask the user to identify the correct query schema by showing them the query results from two query schemas. In this section, we first introduce the method to compute the invalid value and then present our algorithm to modify the database.

Given a database-result pair (D, R), a query schema qs and a selection attribute A, to compute the invalid value, we first identify the *possible valid range* for A with respect to

qs and D, denoted by $PossibleValid_D(A, qs)$. Specially, if qs is the target query schema, then $PossibleValid_D(A, qs)$ satisfied the following property: for each tuple $t \in J_{qs}D$, if $t.A \notin PossibleValid_D(A, qs)$, then t is guaranteed to be a negative tuple. Note that $PossibleValid_D(A, qs)$ is defined to contain all valid-(qs, A) values including possibly some invalid-(qs, A) value. Thus, if $v \notin PossibleValid_D(A, qs)$, then v is an invalid-(qs, A) value. The reason for adopting this approximate definition is that it is amenable to efficient computation.

The $PossibleValid_D(A, qs)$ can be efficiently derived from the selection predicates of Q^{qs}_{min}. Given a query schema qs, Q^{qs}_{min}'s selection predicates with attribute A is of the form $A \in [l, u]$ where $A \in SA$ and $[l, u]$ is the value range of A. Then $PossibleValid_D(A, qs)$ is given by $[\ell', u']$, where $\ell' \leq \ell$ and $u' \geq u$ such that the following two properties hold: (1) if query Q is derived from Q^{qs}_{min} by changing A's selection predicate to $A \in [\ell', u']$, then $Q(D) - R = \emptyset$; and (2) if Q is derived from Q^{qs}_{min} by changing A's selection predicate to $A > u$' or $A < \ell'$, then $Q'(D) - R \neq \emptyset$. Once we calculate $PossibleValid_D(A, qs)$, any value v that $v \in domain(A) - PossibleValid_D(A, qs)$ is an invalid-(qs, A) value.

As each query schema has its own minimal query, for the same attribute A, the possible value range for different query schemas could be different. It is not efficient if we calculate the possible valid range for each query schema to compute the invalid value. Here is an example.

Example 4.3. *Consider three candidate query schemas qs_1, qs_2 and qs_3, and both qs_1 and qs_2 have selection attribute A, where qs_3 does not. To identify whether the target query schema's selection attribute contains A, one naive method is to determine A's possible value range for qs_1 and qs_2 respectively, and then modify A's value to violate both $PossibleValid_D(A, qs_1)$ and $PossibleValid_D(A, qs_2)$. However, it is possible that such a value does not exist if $PossibleValid_D(A, qs_1) \cup PossibleValid_D(A, qs_2) = domain(A)$. In that case, we need to modify database to distinguish qs_1 and qs_3 first, then distinguish qs_2 and qs_3.*

As shown in the example, to identify whether an attribute A is in the target query schema, the naive method needs to compute A's possible value range for each query schema individually. To make the approach efficient, instead of calculating the possible value range for each query schema, we want to find the maximal possible valid value range which is not $domain(A)$ for each attribute A. In order to find the maximal possible value range, we first introduce two lemmas.

Lemma 4.3. *If a query schema qs is a minimal query schema with selection attributes SA, given any query schema qs' where qs' is the same as qs except that $SA' \supset SA$, for each attribute $A' \in SA' - SA$, the $PossibleValid_D(A', qs') = domain(A')$.*

Proof. Because qs is a minimal query schema, $Q^{qs}_{min}(D) - Q(D) = \emptyset$. According to our method to construct the minimal query, for attribute $A \in (SA \cap SA')$, the selection predicates are the same in Q^{qs}_{min} and $Q^{qs'}_{min}$. For an attribute $A' \notin SA$, if $PossibleValid_D(A', qs')$ is $[\ell, u]$, which is a subset of the $domain(A')$, then we get a tuple t in negative partition whose A' value is larger than u or smaller than ℓ. At the same time, for attribute A_i where $A_i \in (SA \cap SA')$, t's value still satisfy the selection predicates of $Q^{qs'}_{min}$, which are the same in Q^{qs}_{min}. This implies that $t \in Q^{qs}_{min}(D)$, contradicting the fact that $Q^{qs}_{min}(D) - Q(D) = \emptyset$. Therefore, the $PossibleValid_D(A', qs') = domain(A')$. □

Lemma 4.4. *Given two minimal query schemas qs_1 and qs_2, which are the same except the selection-attribute sets, denoted by SA_1 and SA_2 respectively, if attribute $A \in (SA_1 \cap SA_2)$, then for query schema qs_3, which is the same as qs_1 and qs_2 except the selection attributes $SA_3 = SA_1 \cup SA_2$, $PossibleValid_D(A, qs_3) \supseteq (PossibleValid_D(A, qs_1) \cup PossibleValid_D(A, qs_2))$.*

Proof. We consider the query schema qs_1 and qs_3 first, where $SA_1 \subset SA_3$. For each common attribute in SA_1 and SA_3, the selection predicates in minimal query are the same. As there are more selection attributes in $Q^{qs_3}_{min}$ than in $Q^{qs_1}_{min}$, it is clear that except attribute A, $Q^{qs_3}_{min}$'s whole selection predicates are more restrictive than $Q^{qs_1}_{min}$'s. Thus, when

we compute the $PossibleValid_D(A, qs_3)$, it is obvious that $PossibleValid_D(A, qs_3) \supseteq PossibleValid_D(A, qs_1)$. Similarly, $PossibleValid_D(A, qs_3) \supseteq PossibleValid_D(A, qs_2)$. Therefore, the lemma holds. □

According to Lemma 4.3, given a query schema qs and attribute A, if $PossibleValid_D(A, qs)$ is not $domain(A)$, then query schema qs', where its $SA' = SA - \{A\}$, is not a minimal query schema. Together with Lemma 4.4, we observe that to find the maximal possible valid range of attribute A, we should compute a query schema $qs*$ satisfying two conditions: (1) the selection attributes $SA*$ should be the superset of selection attributes from all candidate query schemas containing A; (2) the query schema whose selection attributes is $SA * -\{A\}$ is not a minimal query schema. However, sometimes this $qs*$ may not exist. If such a query schema does not exist for A, we have to compute several query schemas for A. We adopt a greedy approach to union the query schemas one by one until the second condition is violated. After that, we get a set of query schemas, and for each one, we compute A's possible valid range individually. If the union of these valid ranges is not $domain(A)$, we use the union as the maximal possible value range. Otherwise, we have to modify the database for each possible valid range before we can identify whether A belongs to the target query schema.

Consider attribute A that partitions candidate query schema $\mathcal{S}$ to two groups $\mathcal{S}_1$ and $\mathcal{S}_2$, where $\mathcal{S}_1$'s selection attributes SA_1 contains attribute A while $\mathcal{S}_2$'s selection attributes SA_2 does not. Suppose the user selects $\mathcal{S}_1$ as the group contains the correct query schema, then all the query schemas in $\mathcal{S}_2$ are not correct. Thus, if attribute A' is in $SA_2 - SA_1$, then A' is not a selection attribute in the target query schema. Intuitively, we can skip asking the question about A'. Furthermore, with the removal of the query schemas in $\mathcal{S}_2$, we reduce the number of the query schemas when we calculate possible value range for other attributes. Similarly when the user selects $\mathcal{S}_2$, all the candidates in $\mathcal{S}_1$ will be eliminated. As the user's selection is unknown, to be conservative, we assume the user always select the group with larger number of candidates. To quantify the effect, we define the *impact*

score of attribute A, denoted by $iscore_1(A)$, as the number of such attributes that are in one candidate group but not the other one, i.e. $iscore_1(A) = min(|SA_2 - SA_1|, |SA_1 - SA_2|)$. $iscore(A)$ indicates the number of iterations we will save. We also define another score $iscore_2(A) = min(|\mathcal{S}_1|, |\mathcal{S}_2|)$, which indicates the number of eliminated candidates.

To optimize the efficiency, before calling the function `Database-Generator`, we first sort the attributes by $iscore_1, iscore_2$ in non-increasing order, and then choose the first one to modify the database. Note that the attribute order may be different in each iteration as the candidate query schemas change, we have to recompute $iscore_1$ and $iscore_2$ in the beginning of each iteration.

Now we present the algorithm as shown in Algorithm 4.3.

Algorithm 4.3: Database-Generator

Input: Database D, candidate query schema set $\mathcal{S}$
Output: A modified database D'

1 **foreach** attribute $A \in \mathcal{S}$'s SA **do**
2   compute $iscore_1(A), iscore_2(A)$
3 Sort attributes by $iscore_1, iscore_2$ in non-increasing order and pick the first attribute A
4 Initialize $atts = \emptyset$, $\mathcal{S}_A = \emptyset$, $QSset = \emptyset$
5 $\mathcal{S}_A = \{qs \in \mathcal{S} | A \in qs$'s SA$\}$
`// all the query schemas containing` A
6 **foreach** query schema $qs_i \in \mathcal{S}_A$ **do**
7     $atts = qs_i$'s SA
8   **foreach** query schema $qs_j \in \mathcal{S}_A$ **do**
9   **if** query schema with SA = $atts \cup qs_j$'SA $- \{A\}$, is not a candidate query schema **then**
10             $atts = atts \cup qs_j$'SA
11   remove qs_j from $\mathcal{S}_A$
12   construct query schema qs whose SA $= atts$
13     $QSset = QSset \cup \{qs\}$
14 Let value range $MaxV = \emptyset$
15 **foreach** query shcema $qs \in QSset$ **do**
16   Compute $PossibleValid_D(A, qs_j)$
17   **if** $MaxV \cup PossibleValid_D(A, qs_j) \neq domain(A)$ **then**
18         $MaxV = MaxV \cup PossibleValid_D(A, qs_j)$
19 Pick value $v \notin MaxV$
20 Modify any positive tuple $t \in D$ to $t' \in D'$ by setting A's value to v
21 **return** D'

Given a set of candidate query schemas, we first calculate the impact score for each se-

lection attribute, and sort the attributes by $iscore_1, iscore_2$ (lines 1 to 3). With the first attribute A, we first find all the candidate query schemas $\mathcal{S}_A$ whose selection attributes contain A (line 5). For each query schema containing A, we union its SA with other query schema's SA. We find the largest attribute set $atts$, such that any query schema with $atts - \{A\}$ as selection attributes is not a potential minimal query schema (lines 6 to 11). Once the query schema is united with others, it's removed from $\mathcal{S}_A$ (line 11). Then we compose a new query schema qs with $atts$ (line 12, 13). For each new query schema, we compute A's possible valid range and find the maximal value range $MaxV$ which is not $domain(A)$ (lines 15 to 18). Then we select a positive tuple from $J_{qs}(D)$ that generates some output tuple $r \in R$. We modify the database tuple $t \in D$ to t' by modifying the value of attribute A such that $t'.A \notin MaxV$ (lines 19 to 21). Then we return the modified database D'.

The complexity of the algorithm is $O(MN^2)$, where M is the number of the selection attribute in candidate query schemas $\mathcal{S}$, and N is the number of candidate query schemas, i.e. $|\mathcal{S}|$.

4.3.3 Result Feedback

Given the modified database D', we highlight the difference between original database D and D' and seek the user's feedback on the following question: If the tuple $t \in D$ is modified to $t' \in D'$ by changing attribute A's value, is $r \in Q(D - \{t\} \cup \{t'\})$? If the user answers "no", then attribute A is contained in target query schema, we choose the group of candidate query schemas whose selection attributes contain attribute A as the candidate query schema for another iterations. Otherwise, we choose the other group of candidates for another iteration.

Note that, it is possible that all the candidate query schemas are not correct. In this case, the select group is an empty set. As shown in Algorithm 4.1, we will pick another valid

join schema or projection attributes to compute the query schema.

4.4 Handling the Scenario Without Positive Partition

As discussed in Section 4.2, we partition tuples in $J_{qs}(D)$ into three subsets, positive, free and negative partitions, where $J_{qs}(D)$ denotes the result of joining all the relations in query schema qs with respect to D. So far, we have introduced an approach to help the user identify the target query schema when the positive partition is not empty. The approach presented in Section 4.3 requires to construct minimal queries, and the selection predicate for each selection attribute is determined by the minimal and maximal value in the positive partition. However, the approach is not applicable if there are only free and negative partitions in $J_{qs}(D)$, as we can not construct minimal query as before. In this section, we discuss how to find the target query schema, if there are no positive partitions in the given dataset. There are two types of SPJ queries we consider. The first one is SPJ queries with set semantics and the second one is the queries with bag semantics. For each query type, we propose an approach to find the target query schema.

As shown in Algorithm 4.1, there are mainly two steps to identify the target query schema. First, we compute the candidate query schemas $\mathcal{S}$, and then we modify the database to partition the candidates and ask the user to pick the correct one. Recall that the key to compute the candidates is to construct a minimal query, which can be used to test whether a given query schema is valid, using positive partitions. However, if positive partition does not exist, it is not clear what tuples would generate the query result.

Example 4.4. *Consider the following database-result pair (D, R), where D consists of a single table. As shown in Table 4.2, there are only two records in the query result, which can partition database D into three partitions. Tuple set $\{E1, E2, E5\}$ is the free partition for the first record in result, and the set $\{E3, E4\}$ is the free partition for the second*

Employee

Eid	name	gender	dept	salary
E1	Alice	F	Sales	4700
E2	Bob	M	IT	4700
E3	Caleb	M	Service	5000
E4	Darren	M	IT	5000
E5	Elly	F	IT	4700
E6	Frank	M	Sales	4900

Database D

salary
4700
5000

Result R

Table 4.2: Employee database and result pair

record. Tuple $E6$ is the only tuple in negative partition. If the target query is a query with bag semantics, i.e., duplicates are allowed in the query result, there could be 6 different combinations of tuples to produce the same query result as R. If the query is under set semantics, i.e., no duplicates occur in the query result, there are more possible tuple combinations to generate the query result, as multiple tuples could be used to generate one tuple result with set semantics.

From the example we can find that without positive partition, it is difficult to identify which tuple is used to generated the query result. As shown, for different query semantics, the tuples we need to generate the same query result could be different.

To help explain our approach, we introduce the notion of *result cover*, which is utilized to compute the candidate query schema. Given a database-result pair (D, R) and query schema qs, if we can find a set of tuples T from the joined relation $J_{qs}(D)$, where T's projection values are exactly the same as the query result R, we call T is a *result cover* (r-cover). In example 4.4, if the target query is a bag-semantics query, we can find 6 r-covers, which respectively are tuple sets $\{E1, E3\}$, $\{E2, E3\}$, $\{E5, E3\}$, $\{E1, E4\}$, $\{E2, E4\}$ and $\{E5, E4\}$. Recall that in Section 4.3, we use positive partition to construct minimal query, which helps us test the validity of a given query schema. In this section, since we do not have positive partition, we use r-cover to construct such a query to verify the query schema. We will illustrate the details in Sections 4.4.1 and 4.4.2. In Section 4.4.3, we

propose a heuristic optimization to solve the problem.

To explain our approach clearly, we describe our algorithms for set-semantics and bag-semantics queries separately.

4.4.1 Queries with Bag Semantics

In this section, we propose our approach to find the target query schema if the target query is a bag-semantics query. The main algorithm is as same as Algorithm 4.1. Here, we first discuss how to compute the result covers and use them to generate the candidate query schema $\mathcal{S}$ (Query-Schema-Generator). Then, we present the algorithm to modify the database and get the target query schema (Database-Generator). Once we get the target query schema, the same approach as Section 4.3 is used to generate the candidate queries.

Algorithm Query-Schema-Generator

As mentioned, to get the candidate query schemas $\mathcal{S}$, we need to construct a query to test the validity of a given query schema, and without positive partitions, we use result covers to construct such a query. In this section, we discuss how to compute the result covers and derive a query to test the validity of a given query schema.

Given a database-result pair (D, R), as we assume there is no positive partition, each output tuple can be generated by multiple tuples from the database. Under bag semantics, we allow duplicate tuples in the query result. Therefore, any two duplicate tuples in R must come from different tuples in free partitions. Let m_i denote the number of duplicate output tuples that are to be generated from the tuples in free partition P_i. There are $\binom{m_i}{|P_i|}$ different combinations to generate the m_i duplicate output tuples, where $|P_i|$ is the total number of tuples in free partition P_i. We refer to each combination as *Partition Cover*

of P_i, denoted by P_i*-cover*. Let k denote the number of free partitions. We can get $\prod_{i=1}^{k} \binom{m_i}{|P_i|}$ r-covers in total.

For each r-cover RC, we compute the candidate query schemas as follows. Given a query schema qs, we first construct a minimal qs-query Q^{qs}_{min}. For each selection attribute A in qs's SA, we construct a selection predicate $A \in [\ell, u]$, where $\ell = min\{\pi_A(t_i)|t_i \in RC\}$ and $u = max\{\pi_A(t_i)|t_i \in RC\}$. If $Q^{qs}_{min}(D) - R \neq \emptyset$, qs is not a valid query schema. Otherwise, qs is a candidate query schema verified by r-cover RC.

Note that not every r-cover can guarantee to compose a valid query schema. For example, in Example 4.4, if we choose $\{E1, E3\}$ as a r-cover, we could not construct a query Q such that $Q(D) - R = \emptyset$. Because $E6$'s every attribute value is in the value range of $E1$ and $E3$ (we ignore attributes 'name' and 'Eid' as it does not make sense to modify these two values), we can not find a SPJ query to eliminate $E6$ from $E1$ and $E3$. If we can't construct a valid query based on a given r-cover RC, we say RC is an *invalid r-cover*. Otherwise, RC is a *valid r-cover*.

Recall that, to compute r-cover, we first select one partition cover for each free partition P, then multiply these partition covers from different free partitions, which could result in a large number of r-covers. For efficiency reasons, we propose an early detection approach to avoid generating the r-covers that are invalid.

Definition 4.4. ***(valid query)*** *Given a database D, a set of free tuples T and query schema $qs = (PA, JS, JP, SA)$, we define qs-query Q as a valid query for T, if $\pi_{PA}(T) \subset Q(D)$ and $Q(D)$ does not contain any negative tuple.*

Lemma 4.5. *Given a database D, a set of free tuples T and query schema qs, if there does not exist a valid query Q for T, then for any free-tuple set $T' \supset T$, there does not exist a qs-query Q' for T', either.*

Proof. If there does not exist a valid query Q for T, that means for any query whose query result contains tuples from T, the query result must also contains at least one negative

tuple. To find a query whose query result contains all tuples from T', where $T' \supset T$, without any negative tuples is impossible. Therefore, there does not exist a qs-query Q' for T' either. $\square$

According to Lemma 4.5, it is clear that given a r-cover RC, if there does not exist a valid query Q for any tuple set T where $T \subset RC$, then there does not exist a valid query for RC. Hence, RC is an invalid r-cover.

To check whether a valid query exists, we first construct a $qs-$query Q in the same way as we construct minimal query. For each selection attribute A, we construct a selection predicate $\{A \in [\ell, u]\}$, where $\ell = min\{\pi_A(t_i)|t_i \in T\}$ and $u = max\{\pi_A(t_i)|t_i \in T\}$. If $Q(D) - R \neq \emptyset$, there does not exist a valid query, otherwise, Q is a valid query.

Lemma 4.6. *Given two sets of free tuples T_1 and T_2, consider a tuple set $T_3 = T_1 \cup T_2$. If there exists a valid query schema qs for T_3, then qs is also a valid query schema for T_1 and T_2.*

Proof. If there exists a valid query schema qs for T_3, then there exists a valid qs-query Q, whose query result contains all tuples of T_3 without any negative tuples. Since $T_3 = T_1 \cup T_2$, Q's query result must also contains tuples from T_1 and T_2. Thus Q is also a valid query for T_1 and T_2, and qs is also a valid query schema for T_1 and T_2. $\square$

Consequently, if query schema qs is a valid query schema for free-tuple set T_1 but not for T_2, then qs is not a valid query schema for T_3, where $T_3 = T_1 \cup T_2$.

Now, we propose our approach to compute r-covers and candidate query schemas. We adopt a bottom-up method to compute r-covers. First, for each free partition P_i, we compute P_i-covers. To compute r-covers, we combine p-covers from different free partitions one by one. Each time we combine a new p-cover from other free partitions, and we examine whether there exists a valid query schema qs for the combined tuples. If not, we do

not combine more p-covers to the current combination (Lemma 4.5). Otherwise, we cache qs to testify the next tuple combination, which unions a new p-cover (Lemma 4.6). When we finally compute a r-cover whose minimal query of query schema qs is also a valid query, qs is a candidate query schema. The whole algorithm is shown in Algorithm 4.4.

Algorithm 4.4: Query-Schema Generator (bag semantics)

```
   Input: join relstion J(D), query resultR
   Output: Candidate query schema S and valid r-covers RC
 1 find all the free partitions {P_1, P_2, ··· , P_n} of J(D)
 2 initialize C[][] = ∅ // store the tuples with valid query in each
       iteration
 3 initialize QS[][] = ∅ // store the query schema in each iteration
 4 foreach free partition P_i(i = 1, 2, ··· , n) do
 5   | ComputeP_i-covers
 6   | foreach T ∈ P_i-covers do
 7   |   | initialize QStmp = ∅
 8   |   | if i == 1 then
 9   |   |   | QStmp = Compute-Valid-Query-Schema(J(D), T)
10   |   | else
11   |   |   | foreach query schema qs ∈ QS[i − 1] do
12   |   |   |   | if valid qs-query Q for T exists then
13   |   |   |   |   | QStmp = QStmp ∪ {qs}
14   |   | if QStmp ≠ ∅ then
15   |   |   | if i == 1 then
16   |   |   |   | C[i].add(T), QS[i].add(QStmp)
17   |   |   | else
18   |   |   |   | foreach T′ ∈ C[i − 1] do
19   |   |   |   |   | Let tuple set Tmp = T ∪ T′
20   |   |   |   |   | initialize tmpqs = ∅
21   |   |   |   |   | foreach qs ∈ QStmp do
22   |   |   |   |   |   | if valid qs-query Q for Tmp exists then
23   |   |   |   |   |   |   | tmpqs = tmpqs ∪ {qs}
24   |   |   |   |   | if tmpqs ≠ ∅ then
25   |   |   |   |   |   | C[i].add(Tmp), QS[i].add(tmpqs)
26 let n be the number of free partitions
27 S = QS[n], RC = C[n]
28 return S, RC
```

As shown in the algorithm, given the query result R and joined relation $J(D)$, we first find all the free partitions by mapping R to $J(D)$ (line 1). For each free partition P_i, we first compute all the P_i-covers (line 5), and then examine whether there exists a valid query schema for each P_i-cover T (lines 8 to 13). If this is the first free partition, we enumer-

Algorithm 4.5: Compute-Valid-Query-Schema

Input: joined relation $J(D)$, free-tuple set T
Output: a set of valid query schema $QStmp$ for T

```
1  Initialize QStmp = ∅, QS_1 = ∅ ;
2  Let AS be the set of all the attributes in the join relation J(D);
3  foreach attribute A ∈ AS do
4  |   Construct query schema qs whose SA = {A};
5  |   if Q^{qs}_{min}(J(D)) − πT ≠ ∅ then
6  |   |   QS_1 = QS_1 ∪ {qs};
7  |   else
8  |   |   QStmp = QStmp ∪ {qs} ;
9  QS_2 = QS_1 ;
10 while QS_2 ≠ ∅ do
11 |   Let QS_3 = ∅;
12 |   foreach qs_2 ∈ QS_2 do
13 |   |   foreach qs_1 ∈ QS_1 do
14 |   |   |   Let atts be the superset of qs_1 and qs_2's selection attributes;
15 |   |   |   if atts has been computed before then
16 |   |   |   |   continue;
17 |   |   |   Construct query schema qs whose SA = atts;
18 |   |   |   if Q^{qs}_{min}(J(D)) − πT ≠ ∅ then
19 |   |   |   |   QS_3 = QS_3 ∪ {qs};
20 |   |   |   else
21 |   |   |   |   QStmp = QStmp ∪ {qs};
22 |   QS_2 = QS_3;
23 return QStmp;
```

ate all the query schemas with function `Compute-Valid-Query-Schema` (line 9). Otherwise, according to Lemma 4.6, we only need to test the valid query schemas from last iteration (line 11). The algorithm of `Compute-Valid-Query-Schema` is shown in Algorithm 4.5, which is similar to Algorithm 4.2. We omit the details of how to test whether a valid query exists, as it is trivial and we already explained it before (line 12). If a valid query schema exists, T will be cached to combine with p-covers from other free partitions (lines 14 to 25). For the first free partition, we simply cache T in C and the valid query schema in QS (line 15, 16). For the subsequent partitions, we combine the p-covers with the cached tuples and examine whether there exists a valid query schema for the new tuple set (lines 19 to 23). Only the tuple sets with valid query schemas are cached for the later iterations. Once we finish combining p-covers from all free partitions, we get the valid r-covers, and the valid query schema corresponding to each r-cover. The complexity of the algorithm is $\prod_{i=1}^{n} C_i \times N$, where C_i is the number of P_i-covers of free partition P_i, and N is the number of query schemas enumerated. With Lemma 4.5 and 4.6, we reduce the number of r-covers and query schemas enumerated as we filter out the invalid ones during the process.

Algorithm Database-Generator

Once we get the candidate query schemas $\mathcal{S}$, we begin modifying the database to identify the target one. One challenge is that each valid query schema may correspond to different r-covers. As each r-cover has different tuples from others, it is possible that there does not exist a tuple shared by all r-covers. As a result, we may not use the same approach as Algorithm 4.3. For example, consider two r-covers containing different tuples, and each of their valid query schema contains selection attribute A. When calculating the maximal possible valid range for attribute A, there may exist conflicts between the value ranges from two r-covers. It is possible that A's invalid value range for the first r-cover is valid for the second r-cover, as they do not use same tuples to generate the query result.

One method to solve the problem is to find a tuple that can be used as positive tuple in Algorithm 4.3, then we can simply reuse the algorithm to modify the database. It is clear that if a tuple appears for all the r-covers, then we can consider it as a positive tuple. Thus, we first partition all the r-covers into different groups on one condition: all r-covers in the same group share at least one common tuple. For each group, as there is at least one common tuple t, it is certain that when we calculate the maximal possible valid range there will be at least one value in common. Therefore, we can avoid the case that attribute A's valid value range in one query schema is another query schema's invalid value range. Therefore, we can adopt the approach in Algorithm 4.3 to distinguish the candidate query schemas. Under the partition condition, there could be multiple ways to partition r-covers. As we need to run Algorithm 4.3 for each partition, to minimize the computation effort, we choose the partitions which result in least number of groups. The algorithm is shown in Algorithm 4.6.

Algorithm 4.6: Database-Generator (bag semantics)

Input: Database D, candidate query schemas $\mathcal{S}$ and valid r-covers RC
Output: User's intended query schema QS

1 $\mathcal{G}$ = `Partition-RCovers(RC)`
2 Sort $\mathcal{G}$ in descending order of $|\mathcal{G}_i|$ and pick the first one $\mathcal{G}_1$
3 let S_{tmp} be an empty set
4 **foreach** r-cover $rc_i \in \mathcal{G}_1$ **do**
5   Let $\mathcal{S}_i$ be the candidate query schema derived from r-cover rc_i
6     $S_{tmp} = S_{tmp} \cup \{\mathcal{S}_j\}$
7 D' =Database-Generator($J(D)$, S_{tmp})
8 **return** D'

We first partition r-covers with function `Partition-RCovers` (line 1). Here we adopt a greedy algorithm which is shown in Algorithm 4.7. We always pick the most frequent tuple t and group the r-covers containing t into one group. Then we sort the groups in the descending order of each group size (line 2). The reason is that the group with more r-covers may have more candidate query schemas, which have higher odds to contain the target query schema. With the largest group $\mathcal{G}_1$, we collect all the candidate query schemas corresponding to each r-cover, and adopt the same approach as Algorithm 4.3 (lines 4 to

Algorithm 4.7: Partition-RCovers

```
   Input: r-covers RC
   Output: A group 𝒢 of r-covers
1  Initialize 𝒢 = ∅;
2  tmpRC = RC;
3  while tmpRC ≠ ∅ do
4      Initilize G = ∅;
5      Let tuple set T be all the tuples in tmpRC;
6      foreach tuple t ∈ T do
7          Initilize g = ∅;
8          foreach rc ∈ tmpRC do
9              if t ∈ rc then
10                 g = g ∪ rc;
11         if |g| > |G| then
12             G = g;
13         remove every rc ∈ G from tmpRC;
14         𝒢 = 𝒢 ∪ {G};
15 return 𝒢 ;
```

6). Note that, we pick the common tuple shared by all r-covers to modify, not the positive tuple as shown in Section 4.3. We return the modified database D' to the user.

Result Feedback

Result Feedback module is same as Section 4.3. Given the modified database D', we highlight the difference between original database D and D' and ask the user to pick the correct query result. Note that, once the user selects the correct group, besides of the candidate query schemas in the group, we also need to collect all the r-covers corresponding to these candidates. Because unlike Section 4.3, Algorithm 4.6 takes both candidate query schemas and corresponding r-covers as input to modify the database.

4.4.2 Queries with Set Semantics

Now we discuss how to find the target query schema when the target query is a set-semantics query.

Comparing to bag semantics, query with set-semantics is more complex since without duplicates in the query result, it is difficult to identify how many tuples are used to generate one tuple in the result. Assume there is one tuple r in query output, and there are m free tuples from database can be projected to r. As the target query is a set-semantics query, to get the result r, there could be $2^m - 1$ different combinations of free tuples, i.e., any non-empty subset of the m tuples could be selected to get the same result r. Given a database-result pair (D, R), where R contains k output tuples, the number of r-covers could be $\prod_{i=1}^{k}(2^{|P_i|-1})$, where P_i is the free partition related to the tuples that can generate r_i in R. Based on the above observation, if a query Q is a valid query, there must exist at least one tuple t_i from each free partition P_i satisfying Q's condition. We refer to this property as at-least-one semantics as addressed in QBO[64]. Due to the at-least-one semantics, the number of r-covers could be very large, and it is clear that enumerating all r-covers to find the candidate query schema is not a practical approach.

Before presenting our approach, we define *minimal r-covers*, and introduce a lemma first.

Definition 4.5. ***(minimal r-cover)*** *Given a database-result pair (D, R), and a r-cover T, if the number of tuples in T is as same as the number of tuples in R, we say T is a minimal r-cover.*

It is clear that a minimal r-cover contains only one tuple from each free partition.

Lemma 4.7. *Given a database-result pair (D, R) and a r-covers T, if qs is a valid query schema for T, then there must exist a minimal r-cover T_m such that qs is also a valid query schema for T_m.*

Proof. (1) If T is a minimal r-cover, then the lemma holds.

(2) Consider when T is not a minimal r-cover. If qs is a valid query schema for T, then there must exist a qs-query Q that $Q(T) = R$, and for each output tuple $r_i \in R$, there must exist a free tuple t_i satisfying query Q. We pick one such tuple from each free partition

and form a minimal r-cover T_m, we have $Q(T_m) = R$. Hence, qs is also a valid query schema for T_m. The lemma holds. □

As Lemma 4.7 shows, to find all the valid query schemas, we only need to examine all the *minimal r-covers*. The total number is $\prod_{i=1}^{k} |P_i|$, which is much less than $\prod_{i=1}^{k}(2^{|P_i|-1})$.

Now we present our algorithm to compute the candidate query schemas. As Lemma 4.5 and 4.6 still holds with set semantics, we adopt the same approach in Algorithm 4.4 to compute the candidates. Recall that in Algorithm 4.4, we first compute p-covers for each free partition and then combine p-covers to compute valid r-covers. Now with set semantics, since we only compute minimal r-cover, for free partition P_i, each free tuple $t \in P_i$ can be considered as a P_i-cover. Thus, we can reuse the Algorithm 4.4 except that we change the p-cover in Algorithm 4.4 to single free tuple. The algorithm is shown in Algorithm 4.8.

Once the candidate query schemas are computed, we can use the same approach in Algorithm 4.6 to modify the database and ask the user to identify the target query schema. The *Result Feedback module* is also as same as in Section 4.4.1, thus we omit the details here.

4.4.3 Heuristic Solution

So far we have presented complete solutions to handle the scenario without positive partition in the database D, both with bag and set semantics. As shown, to compute the candidate query schemas, we need to enumerate all the result covers first. The complexity is quite high as the number of r-covers is quite large. Here we propose a trial-and-error heuristic optimization.

The previous approach in Section 4.4.1 and 4.4.2 requires that we formulate a r-cover which can exactly generate the query result R, which is a very restrict condition. Recall that when computing the candidate query schemas in Section 4.3, we only consider

Algorithm 4.8: Query-Schema Generator (set semantics)

Input: join relstion $J(D)$, query result$R$
Output: Candidate query schema $\mathcal{S}$ and valid r-covers RC

```
1  find all the free partitions {P_1, P_2, ..., P_n} of J(D)
2  initialize C[][] = ∅ // store the tuples with valid query in each
       iteration
3  initialize QS[][] = ∅ // store the query schema in each iteration
4  foreach free partition P_i(i = 1, 2, ..., n) do
5      foreach T ∈ P_i do
6          initialize QStmp = ∅
7          if i == 1 then
8              QStmp = Compute-Valid-Query-Schema(J(D), T)
9          else
10             foreach query schema qs ∈ QS[i − 1] do
11                 if valid qs-query Q for T exists then
12                     QStmp = QStmp ∪ {qs}
13         if QStmp ≠ ∅ then
14             if i == 1 then
15                 C[i].add(T), QS[i].add(QStmp)
16             else
17                 foreach T' ∈ C[i − 1] do
18                     Let tuple set Tmp = T ∪ T'
19                     initialize tmpqs = ∅
20                     foreach qs ∈ QStmp do
21                         if valid qs-query Q for Tmp exists then
22                             tmpqs = tmpqs ∪ {qs}
23                     if tmpqs ≠ ∅ then
24                         C[i].add(Tmp), QS[i].add(tmpqs)
25 let n be the number of free partitions
26 S = QS[n], RC = C[n]
27 return S, RC
```

positive partitions and ignore the free partitions. The reason is it is easy to compute and the target query schema is guaranteed in the candidate query schemas. In addition, the *Database Generator module* only requires to modify the positive tuple. Thus, we do not need free partitions to identify the target query schema.

Now we consider the case without positive partition in the database D, i.e., all the tuples in result R are generated from free tuples. Although we do not know which free tuples contribute to R, we assure that some tuple does. Therefore, for each free tuple t, we can assume that it generates a output tuple in R, and consider t as a positive tuple. Once we have positive tuple, we can reuse the approach in Section 4.3 to identify the target query schema. If our assumption is incorrect, the user should find out that none of the candidate query schema is correct. We can continue to try another free tuple until the user finds the target query schema.

For each free partition, there exists at least one free tuple that contributes to R. Thus, we do not need to enumerate all the free tuples. We only need to examine free tuples from one free partition. To be efficient, we choose the free partition with smallest number of tuples to compute.

4.5 Discussion

In Section 4.2, we give the overview of our approach in Algorithm 4.1. It is clear that for each iteration, all query schemas share the same projection attributes (PA) and join relations (JR). Thus, we assume that all the candidate query schemas are same except their selection attributes (SA) in Section 4.2. In this section, we discuss how to generalize our approach by relaxing the assumption. We first discuss how to handle the query schemas with different PA, then we discuss how to distinguish query schemas with different JR. When we consider query schemas with different PA (or JR), we always assume they

share the same SA, otherwise, we can always change the selection attribute's value to show the differences.

Query schemas with different PA

It is trivial to distinguish two query schemas with different PA. Consider two query schemas qs and qs', where attribute $A \in PA{-}PA'$. As we assume the query schema share the same selection attributes, A is not a selection attribute. Then we choose a positive tuple t to modify. If there is no positive tuple, we can choose a free tuple t which contributes to qs's query result. After modifying t's value of A to a new value, qs's query result will change and qs''s will stay the same. We present the user the difference between two query schemas, and ask him to identify the correct one.

Query schemas with different JR

Without loss of the generality, we assume all the relations are joined under foreign-key relationships. Hence, we do not consider the case two relations can join with different JP. Consider two query schemas qs and qs', where relation $rel \in JA - JA'$. To distinguish the query schema, we modify the join attribute of rel to make sure the tuple can not be joined with others. Given a tuple $t \in rel$ which generates qs's query result r, if we make t can not join with other tuples, then record r will be deleted from qs's query result. However, qs''s result will not be affected. The user can identify the target query schema by looking at the difference in the query results.

4.6 Experimental Study

In this section, we evaluate the usability, efficiency and scalability of our approach using two real datasets. Our experiments were performed on a PC with a Intel Core i7-2600 3.4GHz processor, 8GB RAM, and 320GB SATA HDD running Ubuntu Linux 14.04.

The algorithms were implemented in C++ and the database was managed using MySQL Server 5.5.27.

We first introduce the datasets and test queries in Section 4.6.1. Section 4.6.2 presents the experimental results to show the effectiveness of our approach, in terms of the number of iteration and running time, where the result feedback interactions were returned by a real user choosing the correct result. Section 4.6.3 compares the results of the Schema-based approach (S-QFE) with the Query-based approach (Q-QFE), in terms of the candidate query size. We also conducted a user study with 10 participants in Section 3.6.9, and compare the users' feedback time between S-QFE and Q-QFE to show the usability of our approaches.

4.6.1 Datasets and Queries

We conducted our experiments using two real datasets. The first dataset is a scientific database of biology information taken from SQLShare[1] that consists of two tables: the first table, named PmTE_ALL_DE, contains 3926 records with 16 attributes; and the second table, table_Psemu1FL_RT_spgp_gp_ok, contains 424 records with 3 attributes. We used three queries (denoted by SQ_1, SQ_2 and SQ_3 below) on this database as the target queries. SQ_1 and SQ_3 are real queries posted by some biologist on this dataset. We do not use the same queries in Chapter 3 because so far S-QFE does not support queries with disjunctions.

The second dataset is dataset Adult extracted from the 1994 US Census database[2]. It is a single-relation dataset with 825 tuples. It contains 14 attributes in total. We also used three synthetic queries (denoted by AQ_1, AQ_2 and AQ_3) as target queries to conduct our experiments.

[1] http://escience.washington.edu/sqlshare
[2] http://archive.ics.uci.edu/ml/datasets/Adult

All the target queries are shown in Figure 4.3.

$$
\begin{aligned}
AQ_1 =& \pi_{age,class,occ,edu}(\sigma_{sex=\text{``}F\text{''} \wedge age \geq 64 \\
& \wedge ms=\text{``}Never-married\text{''} \wedge gain>500})adult \\
AQ_2 =& \pi_{age,edu,occ,hour}(\sigma_{occ=\text{``}Farming\text{''} \wedge gain>500 \wedge nc=\text{``}USA\text{''}})adult \\
AQ_3 =& \pi_{age,edu}(\sigma_{occ=\text{``}Tech-support\text{''} \wedge nc=\text{``}USA\text{''}})adult \\
SQ_1 =& \pi_{*}(\sigma_{P.logFC_Fe<0.5 \wedge P.logFC_Fe>-0.5 \wedge P.logFC_P<-1}) \\
& (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok) \\
SQ_2 =& \pi_{*}(\sigma_{P.logFC_Fe \leq -3.61 \wedge P.logFC_Fe>-3.67}) \\
& (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok) \\
SQ_3 =& \pi_{*}(\sigma_{P.logFC_Fe<1 \wedge P.logFC_Si<-1 \\
& \wedge PmTE_ALL_DE.logCPM_Si>1 \wedge P.PValue_P<0.05}) \\
& (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok)
\end{aligned}
$$

Figure 4.3: Test queries for experiments

The number of selection attributes in the six target queries are, respectively, 4, 3, 2, 2, 1 and 4. The cardinalities of the query results for the six target queries are, respectively, 3, 4, 26, 27, 2 and 4 tuples. We generated the initial database-result pairs by executing the above six queries on the database, and we always chose the correct query result as feedback.

4.6.2 Performance of Schema-based Approach

In this section, we present the performance of S-QFE to show the effectiveness. Given the 6 database-result pairs, S-QFE successfully identified the target query schema for 5 queries except AQ_1. As for AQ_1, there are 4 selection attributes in its query schema, but S-QFE only identified 3 attributes without attribute *"sex"*. The reason is that S-QFE found a valid query with the 3 attributes (without "sex") to generate the correct query result. Thus, S-QFE did not add attribute *"sex"* into the query schema as it would be redundant.

	Query No.					
	AQ_1	AQ_2	AQ_3	SQ_1	SQ_2	SQ_3
Total execution time (s)	3.84	6.04	7.21	13.55	2.97	5.80
Time to compute candidates (s)	3.82	6.03	7.18	13.03	2.89	5.72
# of examined query schemas	6475	9721	8438	12274	4152	8295
# of skipped query schemas	9908	6662	7955	53261	61383	57240
# of candidates query schemas $\mathcal{S}$	46	8	5	6	12	82
# of iterations	8	4	4	2	3	6

Table 4.3: Performance for each target query

Table 4.3 shows the following performance statistics: (1) the total running time of S-QFE; (2) the time for computing candidate query schemas; (3) the number of query schemas we examined to find the candidate query schemas; (4) the number of query schemas we skipped when we enumerated all the query schemas to find the candidates. (5) the number of candidate query schemas generated; and (6) the number of iterations to identify the target query schema;

Here the total execution time is the total running time of our approach, which includes the time for mapping projection attributes, partitioning tuples, computing candidate query schemas, modifying database and presenting the new database-result pairs to the user. The time for user's feedback is not included. It is clear that computing the candidate query schemas dominated the whole execution time. The time for all the other operations is less than 0.1 second. More specifically, the time for partitioning tuples is less than 0.01 second, and the time for modifying database in each iteration is less than 1 millisecond. From the user's perspective, the waiting between two iterations is negligible, except for the first iteration, which takes a little longer since S-QFE needs to compute all the candidate query schemas at the beginning.

Computing query schemas takes a long time because S-QFE enumerates all the attribute combinations to find the minimal query schemas. As shown in Table 4.3, overall, the running time increases with the number of examined query schemas. SQ_1 took the longest time, more than 13 seconds, to examine 12274 query schema. Note that AQ_2 had more

query schemas enumerated (9721) than AQ_3 (8438), but it took less time to compute the candidate query schemas. The reason is that AQ_2's query result size is 4, much smaller than AQ_3's, which is 26, and all of them are positive tuples. Recall that to build minimal query for each query schema, we need to check all the positive tuples' values. Therefore, it took longer time to examine one query schema for AQ_3 than AQ_2.

We also present the number of query schemas we skipped when computing candidate query schemas in Table 4.3. To compute the candidates, Algorithm 4.2 requires to enumerate all the query schemas. The number is $2^n - 1$, where n is the number of attributes in dataset. For Adult, the total number is 16383, and it is 65535 for the scientific dataset. However, as our approach stops appending more attributes into candidate query schemas based on Lemma 4.1, we skipped a large number of query schemas to save the computation cost. For Adult dataset, we skipped half of the total query schemas, and more than 80% for the scientific dataset.

Interestingly, although we skipped query schemas because of the candidate query schema, the number of skipped query schemas is not proportional to the number of candidate query schemas. For example, AQ_2 skipped less query schemas than AQ_3 though it had 3 more candidate query schemas. SQ_2 skipped more query schemas than SQ_3 with 70 candidate query schemas less. In fact, more query schemas with smaller set of selection attributes are found, the less of query schemas S-QFE needs to examine. Here we present the number of candidate query schemas generated with different number of selection attributes in Table 4.4. In our experiment, the candidate query schema had 6 selection attributes at most.

As shown in Table 4.4, in Adult dataset, AQ_3 had 5 candidate query schemas with 2 selection attributes, and AQ_2 had 8 candidates with 3 selection attributes. Given a 2-attribute set S_1, the number of attribute sets containing S_1 is 2^{n-2}, where n is the number of total attributes. Thus, if S_1 is a candidate query schema's selection-attribute set, we can skip $2^{n-2} - 1$ query schemas. Similarly, given a 3-attribute set S_2, the number of

# of selection attributes	Query No.					
	AQ_1	AQ_2	AQ_3	SQ_1	SQ_2	SQ_3
1	0	0	0	5	6	5
2	0	0	5	1	6	0
3	14	8	0	0	0	3
4	27	0	0	0	0	38
5	5	0	0	0	0	30
6	0	0	0	0	0	6

Table 4.4: Number of candidate query schemas with different selection attributes size

attribute sets containing S_2 is 2^{n-3}, half size of 2^{n-2}. Therefore, AQ_3 skipped more query schemas. In scientific dataset, note that although SQ_3 had 82 candidate query schemas in total, but only 5 of them had less than 3 attributes. However, SQ_2 had 12 candidate query schemas with 2 or 3 selection attributes. Hence, the number of examined query schema of SQ_2 was only half size of SQ_3.

In terms of iterations, AQ_1 took the most number of iterations with respect to Adult dataset, and SQ_3 took most iterations with respect to the scientific dataset. S-QFE used 8 iterations to find AQ_1 and 6 iterations to find SQ_3. Both of the two queries have the largest number of candidate query schemas among the queries from the same dataset. Both AQ_2 and AQ_3 needed 4 iterations although AQ_2 had 3 more candidate query schemas than AQ_3. Generally speaking, more candidate query schemas requires more iterations to identify the target one.

4.6.3 Comparing Query-based and Schema-based approaches

In this section, we compare the Schema-based approach (S-QFE) with Query-based approach (Q-QFE) in terms of the number of iterations, number of candidate query schemas (candidate queries) and running time to identify the target query. We still use the 6 queries in Figure 4.3 as the target queries to conduct the experiments.

	Query No.					
	AQ_1	AQ_2	AQ_3	SQ_1	SQ_2	SQ_3
Q-QFE QG time	0.34	0.49	0.84	x	1.851	2.282
Q-QFE DG time	2.44	2.38	1.23	x	6.652	5.346
Q-QFE total time	2.78	2.87	2.08	x	8.503	7.628
S-QFE total time	3.84	6.04	7.21	13.55	2.97	5.80

(a) Execution time(in secs)

	Query No.					
	AQ_1	AQ_2	AQ_3	SQ_1	SQ_2	SQ_3
# of candidate queries in Q-QFE	8	5	5	-	9	7
# of candidate query schemas in S-QFE	46	8	5	6	12	82
# of Q-QFE iterations	3	3	2	-	4	3
# of S-QFE iterations	8	4	4	2	3	6

(b) Number of candidate queries and iterations

Query No.	Approach	Iteration No.							
		1	2	3	4	5	6	7	8
AQ_1	S-QFE	3822	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	Q-QFE	1419	1247	106	-	-	-	-	-
AQ_2	S-QFE	6025	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	Q-QFE	1663	1140	65	-	-	-	-	-
AQ_3	S-QFE	7181	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	Q-QFE	2001	56	-	-	-	-	-	-
SQ_2	S-QFE	2886	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	Q-QFE	3632	2125	1764	982	-	-	-	-
SQ_3	S-QFE	5717	< 1	< 1	< 1	< 1	< 1	< 1	< 1
	Q-QFE	4298	1987	1436	-	-	-	-	-

(c) System processing time for each iteration (in milliseconds)

Table 4.5: Results of two approaches

The total execution time of two approaches is shown in Table 4.5.(a). The execution time of Q-QFE approach is the sum of the Query-Generator running time (QG time) and Database-Generator running time (DG time). We do not include feedback time here.

Note that for SQ_1, we use "x" to indicate the execution time of Q-QFE. Because the Query Generator took too much time to generate the candidate queries, we have to terminate the system manually, the experiment failed for SQ_1 query. Thus the number of candidate queries and iterations are also not available, indicated by "-". Besides, as mentioned in Section 4.6.2, S-QFE could not find the original query for AQ_1, but since it could find a

similar query (missing one redundant selection attribute), we just considered that it found the target query. Similarly, for queries AQ_2 and SQ_3, Q-QFE did not generate the original query, but it still found some similar queries as the original query. We also considered that Q-QFE generated the target query.

As shown in Table 4.5.(a), there is no clear winner between the two approaches. For AQ_1, AQ_2 and AQ_3, Q-QFE was faster than S-QFE, and for SQ_2 and SQ_3, S-QFE needed less time. For Q-QFE approach, we can see that Database Generator always took more time than the Query Generator, because in each iteration, it needed to calculate the balance score and the modification cost, then find the best way to partition queries. As to S-QFE approach, as discussed earlier, the running time was dominated by the time to find candidate query schemas, which varies a lot for different queries.

The number of candidate queries generated from Q-QFE, the number of candidate query schemas from S-QFE and the number of iterations are shown in Table 4.5.(b). There is also no clear winner. Generally, there are more candidate query schemas from S-QFE than the candidates from Q-QFE's. Except SQ_2, S-QFE required more iterations than Q-QFE to identify the target one. The reason is that Q-QFE can partition queries into multiple groups, and use balance score to control the balance, while S-QFE can only partition query schemas into two group each time. However, S-QFE only needs to modify one tuple in each iteration.

As for each iteration, except the first iteration, S-QFE took much less time for each iteration, usually less than 1 millisecond. Because at the first iteration S-QFE computed candidate query schemas, which is quite time-consuming. However, Q-QFE usually took around 2 seconds between iterations, since it needed to find the optimal way to partition queries based on the user's feedback. The system processing time for each iteration is shown in Table 4.5.(c). Note that if the target query schema was identified in the kth iteration where $k < 8$, then the timing value for each of the remaining iterations will be indicated by '-'.

Original Data:

id	age	workclass	education	education_num	material_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	native_country
41979	48	Self-emp-not-inc	Prof-school	15	Married-civ-spouse	Sales	Husband	White	Male	0	0	20	United-States
2349	55	Private	Bachelors	13	Married-civ-spouse	Sales	Husband	White	Male	0	0	50	United-States
41104	22	Self-emp-not-inc	Some-college	10	Never-married	Sales	Own-child	White	Male	0	0	20	United-States
44488	18	Private	11th	7	Never-married	Sales	Own-child	White	Female	0	0	5	United-States
10197	32	Private	Some-college	10	Married-civ-spouse	Tech-support	Husband	White	Male	0	0	45	United-States

Original Result:

age	education
71	Bachelors
45	HS-grad
32	Bachelors
25	Bachelors
23	Some-college

start

Original Data:

id	age	workclass	education	education_num	material_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	native_country
41979	48	Self-emp-not-inc	Prof-school	15	Married-civ-spouse	Sales	Husband	White	Male	0	0	20	United-S
2349	55	Private	Bachelors	13	Married-civ-spouse	Sales	Husband	White	Male	0	0	50	United-S

Original Result:

age	education
71	Bachelors
45	HS-grad

At query time:

OLD AND NEW DATABASE

	id	age	workclass	education	education_num	material_status	occupation	relationship	race	sex	capital_gain	capital_loss	hours_per_week	native_country
old	1230	71	Private	Bachelors	13	Divorced	Tech-support	Own-child	White	Female	2329	0	16	United-States
new	1230	71	Private	Prof-school	13	Divorced	Tech-support	Own-child	White	Female	2329	0	16	United-States

Should the new tuple be in query result? Yes No

next

Figure 4.4: User interface screen capture

4.6.4 User Study

In this section, we present the results of a user study conducted with 10 participants (all of whom were CS students) to evaluate the feasibility of our approach. The screen capture of the system user interface is shown in Figure 4.4. Similar to the user interface in Figure 3.14, the system first showed the input database-result pair to the user. The user can scroll up and down to browse the tuples in database and query result. In each iteration, the system highlighted the differences between original and modified tuples. We used

different colors to mark the modified attribute, the old and updated values to help users examine the modifications. To make it easier for users, instead of asking the user to enter the group number of the correct query result (in Section 3.6.9), we provided Yes/No buttons. We asked the question "After the modification, whether the new tuple should be in query result". The user keeps clicking Yes/No buttons until he identifies the target query schema.

For this experiment, we used the Adult relation and three queries in Section 4.6.1 as target queries. This dataset was chosen over the scientific dataset as we felt that its data domain would be easier to understand for users. Before the participants started, we first expressed the query intentions to the participants in written English, rather than the SQL queries, because our purpose is to help users construct SQL queries. For each query, we report the user's feedback time at each iteration, which is shown in Table 4.6 to 4.8. The system execution time is not included. Note that if the target query schema was identified in the kth iteration where $k < 4$, then the timing value for each of the remaining iterations will be indicated by '-'.

First of all, all of the participants could identify all the target queries correctly. As shown in Tables 4.6 to 4.8, Q-QFE always took less iterations to identify the target query. For AQ_1, Q-QFE saved 5 iterations comparing to S-QFE, and for AQ_2 and AQ_3, it saved 1 and 2 iterations respectively. However, the average feedback time at one iteration of Q-QFE was around 18 seconds, which is much longer than the average time of S-QFE, less than 10 seconds. It means that S-QFE requires less effort for users to examine the data examples. Because S-QFE only modifies one tuple each time, and it asks a yes/no question, which is easier to answer.

It took the participants longer time at the beginning for S-QFE. The reason is that the participants needed some time to understand the meaning of the query and the data schema. After they became familiar with the query meaning and the data schema, it only took around 8 seconds for each iteration. On the other hand, there is no such trend for Q-QFE

User	Approach	Iteration No.							
		1	2	3	4	5	6	7	8
1	S-QFE	3.41	3.05	5.09	5.81	2.21	3.28	4.15	4.75
	Q-QFE	10.68	14.73	9.43	-	-	-	-	-
2	S-QFE	12.96	5.82	4.36	2.89	22.37	11.95	5.08	9.97
	Q-QFE	14.67	21.18	8.36	-	-	-	-	-
3	S-QFE	25.53	18. 10	9.51	5.01	2.82	6.97	6.93	4.98
	Q-QFE	14.93	25.59	10.57	-	-	-	-	-
4	S-QFE	4.18	2.52	5.34	2.90	9.17	4.86	5.98	4.81
	Q-QFE	9.74	18.42	12.87	-	-	-	-	-
5	S-QFE	10.81	19.92	4.99	5.68	19. 83	2.97	6.56	4.65
	Q-QFE	13.55	29.81	18.45	-	-	-	-	-
6	S-QFE	23.81	18.89	16.04	12.94	29.25	10.31	5.44	10.24
	Q-QFE	37.13	53.82	18.56	-	-	-	-	-
7	S-QFE	19.05	31. 25	13.06	5.03	27.6	8.01	6.92	7.97
	Q-QFE	25.83	50.4	15.8	-	-	-	-	-
8	S-QFE	10.51	20.04	3.39	5.82	47.22	7.18	3.76	5.50
	Q-QFE	9.85	17.73	7.86	-	-	-	-	-
9	S-QFE	11.54	10.67	5.02	6.29	32.33	11.45	8.24	4.93
	Q-QFE	18.91	61.58	10.81	-	-	-	-	-
10	S-QFE	11.54	9.71	5.49	6.06	19.58	6.62	6.34	11.20
	Q-QFE	19.22	16.49	13.27	-	-	-	-	-

Table 4.6: Feedback time for AQ_1 (in secs)

approach, because unlike S-QFE, Q-QFE could modify more than 1 tuples in one iteration. Thus, the user's feedback is more related to the modifications in each iteration. For S-QFE, in some iteration, it took a little longer for the participants to identify the query, for example, the 5th iteration of AQ_1. The reason is that they were confused by the attribute name, like *"relationship"* and *"material_status"*. In AQ_1, the selection condition is *material_status = Never-married*. In the 5th iteration, we modified the tuple by changing its $relationship$ value to "Unmarried", which was a little ambiguous. If the user is familiar with the dataset, he would be aware of the problem.

Now we compare the total execution time between Q-QFE and S-QFE, including both system running time and user's feedback time. The results are shown in Figure 4.5. We also present the average time of each iteration in Table 4.9 to 4.11.

For AQ_1, nine participants spent less time to identify the target query with Q-QFE than

User	Approach	i-th iteration			
		1	2	3	4
1	S-QFE	8.58	6.43	5.98	1.76
	Q-QFE	9.13	12.19	8.97	-
2	S-QFE	11.82	8.27	10.04	6.21
	Q-QFE	14.98	16.23	12.76	-
3	S-QFE	6.79	5.56	6.02	5.03
	Q-QFE	3.38	12.34	12.45	-
4	S-QFE	5.02	4.17	4.84	3.26
	Q-QFE	11.21	9.45	10.93	-
5	S-QFE	10.61	10.55	10.05	3.94
	Q-QFE	8.65	10.92	9.67	-
6	S-QFE	18.44	11.63	6.12	4.82
	Q-QFE	28.37	14.09	10.82	-
7	S-QFE	11.35	11.17	19.94	4.63
	Q-QFE	17.23	12.64	13.18	-
8	S-QFE	7.39	18.74	10.13	6.01
	Q-QFE	10.21	11. 95	11.84	-
9	S-QFE	5.27	5.01	5.42	5.40
	Q-QFE	11.47	12.02	10.79	-
10	S-QFE	12.37	6.68	6.19	7.41
	Q-QFE	10.79	6.73	8.08	-

Table 4.7: Feedback time for AQ_2 (in secs)

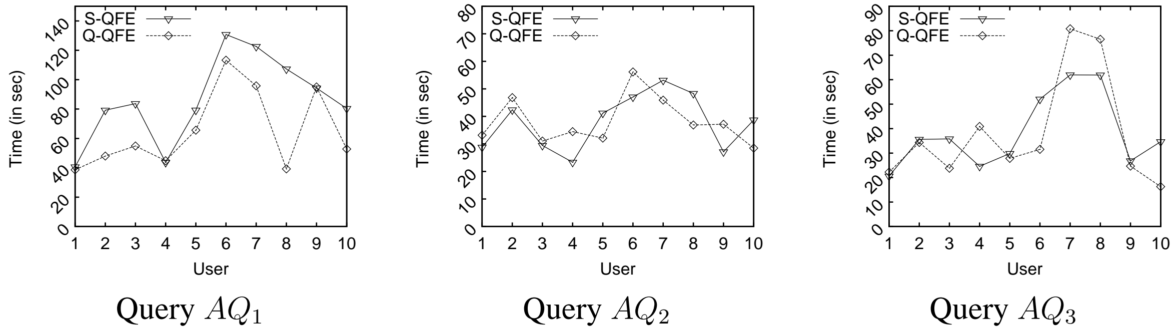
S-QFE. It took user 9 almost the same time using Q-QFE (95.08 seconds) and S-QFE (94.31 seconds). The reason is that S-QFE required 5 more iterations to find the target query schema. As for AQ_2 and AQ_3, half of the participants found it was faster to use S-QFE approach, while the other half took less time to identify the query with Q-QFE approach. Overall, two approaches are comparable.

In terms of interaction time, for AQ_1, the longest feedback time in Q-QFE is 61 seconds, and the shortest is 7.8 seconds. With S-QFE, the longest time is 47 seconds and the shortest is 2.2 seconds. ForAQ_2, the longest and shortest feedback time is 18 and 1.7 seconds in S-QFE, and 28.4 and 3.9 seconds in Q-QFE. As for AQ_3, the longest and shortest feedback time is 39 second and 2.7 second in S-QFE and 52 and 8.5 seconds in Q-QFE. Also as shown in Table 4.9 to 4.11, the average time of S-QFE at each iteration is also much less than Q-QFE.

User	Approach	i-th iteration			
		1	2	3	4
1	S-QFE	2.88	3.05	4.62	2.93
	Q-QFE	10.62	9.53	-	-
2	S-QFE	12.55	4.69	5.82	5.36
	Q-QFE	18.45	13.82	-	-
3	S-QFE	15.31	7.84	2.93	2.52
	Q-QFE	10.11	11.62	-	-
4	S-QFE	7.13	3.96	2.59	3.71
	Q-QFE	20.88	17.93	-	-
5	S-QFE	9.84	5.38	2.73	4.68
	Q-QFE	17.64	8.13	-	-
6	S-QFE	21.66	9.84	6.87	6.47
	Q-QFE	15.49	13.96	-	-
7	S-QFE	32.22	12.13	3.70	6.68
	Q-QFE	52.81	25.92	-	-
8	S-QFE	38.97	5.39	4.05	6.26
	Q-QFE	48.24	26.21	-	-
9	S-QFE	3.96	3.43	5.41	6.81
	Q-QFE	14.05	8.55	-	-
10	S-QFE	6.45	5.98	3.14	11.89
	Q-QFE	7.84	6.38	-	-

Table 4.8: Feedback time for AQ_3 (in secs)

There is no clear trend which approach is better. In general, Q-QFE needs less iterations to identify the target query, but at each iteration it takes the user longer time to examine the examples comparing with S-QFE. As a result of our analysis, when Q-QFE generates more candidate queries and the query schema contains many attributes, it takes Q-QFE more time to compute the optimal way to modify database, and because of too many modifications at one iteration, it takes the user longer time to examine the modified database. On the contrary, it is not suitable to use S-QFE approach when the candidate queries are few, as S-QFE takes a lot of time when calculating candidate query schemas by enumerating all the selection-attribute sets.



Query AQ_1 Query AQ_2 Query AQ_3

Figure 4.5: Total time to find target query (in secs)

4.7 Conclusion

In this chapter, we have proposed a Schema-based approach of QFE (S-QFE) to help the user construct queries. S-QFE takes as input an initial user-specified pair of sample database D and output table R for the user's target query on D, and outputs candidate queries with the correct query schema as the user's target query. Unlike Q-QFE, we propose a novel algorithm to help users identify the valid query schema first through a sequence of iterations with the user to obtain the feedback on the correct query result on modified input database. S-QFE does not expect users to be familiar with SQL and only requires that users are able to determine whether a given output table is the result of his or her intended query on a given input database.

Our experimental evaluation demonstrates the feasibility of our approach and the efficiency of our techniques. We also conduct a user study to show the effectiveness. The results show that our approach is easy to use. And the comparison between Q-QFE and S-QFE also demonstrates that two approaches are comparable.

As part of future work, we plan to generalize our approach to handle a larger class of queries, such as SPJ-union query, query with aggregation function, etc. We would also like to further integrate Q-QFE and S-QFE into a hybrid system, and build an accurate cost estimate model to adopt the proper approach.

User	Approach	Total time	# of iterations	Average time
1	S-QFE	40.59	8	5.07
	Q-QFE	38.62	3	12.87
2	S-QFE	79.24	8	9.91
	Q-QFE	47.99	3	16.00
3	S-QFE	83.69	8	10.46
	Q-QFE	54.87	3	18.29
4	S-QFE	43.60	8	5.45
	Q-QFE	44.83	3	14.94
5	S-QFE	79.25	8	9.91
	Q-QFE	65.59	3	21.86
6	S-QFE	130.76	8	16.35
	Q-QFE	113.29	3	37.76
7	S-QFE	122.64	8	15.33
	Q-QFE	95.81	3	31.94
8	S-QFE	107.26	8	13.41
	Q-QFE	39.22	3	13.07
9	S-QFE	94.31	8	11.79
	Q-QFE	95.08	3	31.69
10	S-QFE	80.38	8	10.05
	Q-QFE	52.76	3	17.59

Table 4.9: Time results of Q-QFE and S-QFE forAQ_1 (in secs)
.

User	Approach	Total time	# of iterations	Average time
1	S-QFE	28.79	4	7.20
	Q-QFE	33.16	3	11.05
2	S-QFE	42.38	4	10.59
	Q-QFE	46.84	3	15.61
3	S-QFE	29.44	4	7.36
	Q-QFE	31.04	3	10.35
4	S-QFE	23.33	4	5.83
	Q-QFE	34.46	3	11.49
5	S-QFE	41.19	4	10.29
	Q-QFE	32.11	3	10.70
6	S-QFE	47.05	4	11.76
	Q-QFE	56.15	3	18.72
7	S-QFE	53.13	4	13.28
	Q-QFE	45.92	3	15.31
8	S-QFE	48.31	4	12.08
	Q-QFE	36.87	3	12.29
9	S-QFE	27.14	4	6.79
	Q-QFE	37.15	3	12.38
10	S-QFE	38.69	4	9.67
	Q-QFE	28.47	3	9.49

Table 4.10: Time results of Q-QFE and S-QFE forAQ_2 (in secs)

.

User	Approach	Total time	# of iterations	Average time
1	S-QFE	20.59	4	5.15
	Q-QFE	22.23	2	11.12
2	S-QFE	35.63	4	8.91
	Q-QFE	34.35	2	17.17
3	S-QFE	35.81	4	8.95
	Q-QFE	23.81	2	11.91
4	S-QFE	24.60	4	6.15
	Q-QFE	40.89	2	20.45
5	S-QFE	29.87	4	7.47
	Q-QFE	27.85	2	13.79
6	S-QFE	52.05	4	13.01
	Q-QFE	31.53	2	15.76
7	S-QFE	61.94	4	15.49
	Q-QFE	80.81	2	40.41
8	S-QFE	61.88	4	15.47
	Q-QFE	76.53	2	38.23
9	S-QFE	26.82	4	6.71
	Q-QFE	24.68	2	12.34
10	S-QFE	34.67	4	8.67
	Q-QFE	16.30	2	8.15

Table 4.11: Time results of Q-QFE and S-QFE forAQ_3 (in secs)
.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

In this thesis, aiming to help non-expert database users construct SQL queries, we propose a novel approach called Query from Examples (QFE), which is designed for users who might be unfamiliar with SQL, and only requires that the user is familiar with the dataset and able to determine whether a given output table is the result of his or her intended query on a given input database. The user inputs a sample database D and an output table R which is the result of the his/her intended query Q on D, QFE will first generate a set of candidate queries or query schemas, and then help the user to identify the target query from these candidates by adopting an instance-driven interactive method.

5.1 Contributions

In this thesis, we first introduced Query-based approach of QFE (Q-QFE). We adopted an interactive instance-driven approach to partition candidate queries into different subsets with different query results. By using data examples, our system is quite straightforward and user friendly. We analyzed the characteristics that a good data example should satisfy and proposed an algorithm to derive it. To make the system more practical, we also proposed a novel cost model to estimate the user's workload, so as to minimize the user's effort to identify the target query. Besides, we also conducted an extensive experimental study over real datasets and user studies, which showed that our system is effective and efficient.

Secondly, we designed a Schema-based approach of QFE (S-QFE). Given a sample database D and an output table R as input, our approach first identifies the target query schema, and then generates a set of candidate queries sharing the target query schema, which can transform D to R. We introduced a novel method to help the user identify the target query schema through a sequence of iterations with the user to provide feedback on the correct query result on a modified input database. By involving user to the process of query derivation, we can filter out the incorrect query schemas in advance, and reduce the search space. An experimental study over different datasets was also conducted to show that S-QFE is efficient. We also conduct a user study to show the effectiveness of our approach.

5.2 Future Work

There are several possible directions to extend QFE.

First, we would like to extend QFE to handle more queries types including SPJ-union (SPJU) queries, group-by aggregation (SPJA) queries, etc.

Second, for Q-QFE, to reduce users' waiting time, we can use parallelization techniques to speed up the system performance. One method is to pipeline the two components, such that once the *Candidate Generator module* starts generating queries, we run the *Database Generator module* immediately while the *Candidate Generator module* continues generating more queries. Another method is to use multiple threads to execute tasks i.e. partitioning queries in parallel. Moreover, we can take advantage of users' feedback to filter the queries at the beginning that can be ensured not useful for users.

Third, we would like to integrate Q-QFE and S-QFE to build a hybrid system. If we can directly generate a small number of candidate queries by query generator, we do not adopt S-QFE to identify query schema. Otherwise, we adopt S-QFE to identify query schema first. In addition, to be more flexible, we would like to provide an option that the system can terminate S-QFE anytime, and use the remaining candidate query schemas to generate candidate queries and adopt Q-QFE to identify the target query. The system could estimate the user's effort accurately and decide which approach is more efficient. To sum up, S-QFE and Q-QFE could be easily switch in order to reduce the user's effort.

Another possible direction is to extend this work to handle incomplete query results. It is common that users may not know the full query results even for a database that he is familiar with, or the full query result may be large such that users are reluctant to completely specify. Thus it is important and useful to generate the user's intended query when given a database and part of the query result.

In addition, we would also like to explore other ways to specify the input data. For example, a user only needs to input a set of keywords, and then the system will automatically generate a small set of data (sampling from the existing database) to let users mark the query result they want. It would also be useful to conduct a user study to examine how the size of the data the system automatically generates affects the queries that our query generator produces and how the size could possibly reduce/enlarge the partition space.

BIBLIOGRAPHY

[1] Sloan digital sky survey. http://www.sdss.org/.

[2] S4: Top-k Spreadsheet-Style Search for Query Discovery. ACM Association for Computing Machinery, June 2015.

[3] Azza Abouzied, Dana Angluin, Christos Papadimitriou, Joseph M. Hellerstein, and Avi Silberschatz. Learning and verifying quantified boolean queries by example. In Proceedings of the 32Nd Symposium on Principles of Database Systems, 2013.

[4] Azza Abouzied, Joseph Hellerstein, and Avi Silberschatz. Dataplay: Interactive tweaking and example-driven correction of graphical database queries. In Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology, 2012.

[5] Javad Akbarnejad, Gloria Chatzopoulou, Magdalini Eirinaki, Suju Koshy, Sarika Mittal, Duc On, Neoklis Polyzotis, and Jothi S. Vindhiya Varman. Sql querie recommendations. Proc. VLDB Endow., 3:1597–1600, 2010.

[6] Bogdan Alexe, Laura Chiticariu, Renée J. Miller, and Wang Chiew Tan. Muse: Mapping understanding and design by example. In ICDE, 2008.

[7] Bogdan Alexe, Laura Chiticariu, and Wang-Chiew Tan. Spider: A schema mapping debugger. In VLDB, 2006.

[8] Carsten Binnig et al. Qagen: generating query-aware test databases. In SIGMOD, pages 341–352, 2007.

[9] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In ICDE, pages 506–515, 2007.

[10] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. In Transactions on Knowledge and Data Engineering. IEEE Computer Society, 2006.

[11] Huanhuan Cao, Daxin Jiang, Jian Pei, Qi He, Zhen Liao, Enhong Chen, and Hang Li. Context-aware query suggestion by mining click-through and session data. In SIGKDD, pages 875–883, 2008.

[12] Ugur Çetintemel, Mitch Cherniack, Justin DeBrabant, Yanlei Diao, Kyriaki Dimitriadou, Alexander Kalinin, Olga Papaemmanouil, and Stanley B. Zdonik. Query steering for interactive data exploration. In CIDR, 2013.

[13] Adriane Chapman and H. V. Jagadish. Why not? In SIGMOD, pages 523–534, 2009.

[14] Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. Query recommendations for interactive database exploration. In SSDBM, 2009.

[15] Gloria Chatzopoulou et al. The querie system for personalized query recommendations. IEEE Data Eng. Bull., 34(2):55–60, 2011.

[16] Rada Chirkova. Equivalence and minimization of conjunctive queries under combined semantics. In ICDT, pages 262–273, 2012.

[17] W. W. Chu and Q. Chen. A structured approach for cooperative query answering. IEEE Trans. on Knowl. and Data Eng., 6(5):738–749, October 1994.

[18] Sara Cohen. Equivalence of queries that are sensitive to multiplicities. VLDB J., 18(3):765–785, 2009.

[19] Sara Cohen et al. Equivalences among aggregate queries with negation. ACM Trans. Comput. Logic, 6(2):328–360, 2005.

[20] Sara Cohen et al. Deciding equivalences among conjunctive aggregate queries. J. ACM, 54(2), 2007.

[21] Jonathan Danaparamita and Wolfgang Gatterbauer. Queryviz: helping users understand sql queries and their patterns. In EDBT, pages 558–561, 2011.

[22] David DeHaan. Equivalence of nested queries with mixed semantics. In PODS, pages 207–216, 2009.

[23] E. Demidova, Xuan Zhou, and W. Nejdl. A probabilistic scheme for keyword-based incremental query construction. Knowledge and Data Engineering, IEEE Transactions on, 24(3):426–439, March 2012.

[24] Elena Demidova, Xuan Zhou, and Wolfgang Nejdl. Efficient query construction for large scale data. In SIGIR, pages 573–582, 2013.

[25] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In SIGMOD, SIGMOD '14, pages 517–528, 2014.

[26] Ronald Fagin and Moshe Y. Vardi. Armstrong databases for functional and inclusion dependencies. Inf. Process. Lett., 16(1):13–19, 1983.

[27] Carles Farré, Ernest Teniente, and Toni Urpí. Checking query containment with the cqc method. Data Knowl. Eng., 53(2):163–223, 2005.

[28] Terry Gaasterland, Parke Godfrey, and Jack Minker. An overview of cooperative answering. J. Intell. Inf. Syst., 1(2):123–157, 1992.

[29] Wolfgang Gatterbauer. Databases will visualize queries too. PVLDB, 4(12), 2011.

[30] Arnaud Giacometti, Patrick Marcel, Elsa Negre, and Arnaud Soulet. Query recommendations for olap discovery driven analysis. In Proceedings of the ACM twelfth international workshop on Data warehousing and OLAP, DOLAP '09, pages 81–88, 2009.

[31] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Exploiting constraint-like data characterizations in query optimization. In SIGMOD, pages 582–592, 2001.

[32] Jiafeng Guo, Xueqi Cheng, Gu Xu, and Huawei Shen. A structured approach to query recommendation with social annotation data. In CIKM, pages 619–628, 2010.

[33] Melanie Herschel and Mauricio A. Hernández. Explaining missing answers to spjua queries. PVLDB, 3(1-2):185–196, 2010.

[34] Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. Artemis: A system for analyzing missing answers. Proc. VLDB Endow., 2(2):1550–1553, August 2009.

[35] Bill Howe, Garret Cole, Nodira Khoussainova, and Leilani Battle. Automatic starter queries for ad hoc databases. In SIGMOD'11: Proc. of the ACM SIGMOD Int. Conf. on Management of Data (demo), 2011.

[36] Bill Howe, Garrett Cole, Emad Souroush, Paraschos Koutris, Alicia Key, Nodira Khoussainova, and Leilani Battle. Database-as-a-service for long-tail science. In SSDBM, 2011.

[37] Jiansheng Huang et al. On the provenance of non-answers to queries over extracted data. PVLDB, 1(1):736–747, 2008.

[38] Yannis E. Ioannidis. From databases to natural language: The unusual direction. In NLDB, pages 12–16, 2008.

[39] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In SIGMOD, pages 13–24, 2007.

[40] T. S. Jayram et al. The containment problem for real conjunctive queries with inequalities. In PODS, pages 80–89, 2006.

[41] Nodira Khoussainova et al. Snipsuggest: Context-aware autocompletion for sql. PVLDB, 4(1):22–33, 2010.

[42] Nodira Khoussainova, YongChul Kwon, Wei-Ting Liao, Magdalena Balazinska, Wolfgang Gatterbauer, and Dan Suciu. Session-based browsing for more effective query reuse. In SSDBM, pages 583–585, 2011.

[43] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In VLDB, pages 199–210, 2006.

[44] Georgia Koutrika, Alkis Simitsis, and Yannis E. Ioannidis. Explaining structured queries in natural language. In ICDE, pages 333–344, 2010.

[45] Alon Y. Levy and Yehoshua Sagiv. Queries independent of updates. In VLDB, pages 171–181, 1993.

[46] Eric Lo, Nick Cheng, and Wing-Kai Hon. Generating databases for query workloads. PVLDB, 3(1):848–859, 2010.

[47] Heikki Mannila and Kari-Jouko Räihä. Automatic generation of test data for relational queries. J. Comput. Syst. Sci., 38(2):240–258, 1989.

[48] Fabien De Marchi, Stéphane Lopes, and Jean-Marc Petit. Efficient algorithms for mining inclusion dependencies. In EDBT, 2002.

[49] Qiaozhu Mei, Dengyong Zhou, and Kenneth Church. Query suggestion using hitting time. In CIKM, pages 469–478, 2008.

[50] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In EDBT, EDBT '09, pages 862–873, 2009.

[51] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In SIGMOD, pages 499–510, 2008.

[52] A Motro. Intensional answers to database queries. Knowledge and Data Engineering, IEEE Transactions on, 6(3):444–454, Jun 1994.

[53] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. Exemplar queries: Give me an example of what you need. PVLDB, 7(5):365–376, 2014.

[54] Ion Muslea and Thomas J. Lee. Online query relaxation via bayesian causal structures discovery. In AAAI - Volume 2, pages 831–836, 2005.

[55] Arnab Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In Proceedings of the 2007 ACM SIGMOD international conference on Management of data, pages 1156–1158, 2007.

[56] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In SIGMOD, pages 245–256, 2009.

[57] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In SIGMOD Conference, pages 1099–1110, 2008.

[58] Li Qian, Michael J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In SIGMOD, 2012.

[59] Anish Das Sarma, Aditya G. Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In ICDT, pages 89–103, 2010.

[60] Shetal Shah et al. Generating test data for killing sql mutants: A constraint-based approach. In ICDE, pages 1175–1186, 2011.

[61] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. Discovering queries based on example tuples. In SIGMOD, pages 493–504, 2014.

[62] Alkis Simitsis and Yannis E. Ioannidis. Dbmss should talk back too. In CIDR, 2009.

[63] Sandeep Tata and Guy M. Lohman. Sqak: Doing more with keywords. In SIGMOD, pages 889–902, 2008.

[64] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In SIGMOD, pages 535–548, 2009.

[65] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query reverse engineering. The VLDB Journal, 23(5), 2014.

[66] Jeffrey D. Ullman. Information integration using logical views. Theor. Comput. Sci., 239(2):189–210, 2000.

[67] Fang Wei and Georg Lausen. Containment of conjunctive queries with safe negation. In ICDT, pages 346–360, 2002.

[68] Fang Wei and Georg Lausen. A unified apriori-like algorithm for conjunctive query containment. In IDEAS, pages 111–120, 2008.

[69] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C. Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. A colorful approach to text processing by example. In UIST, 2013.

[70] Meihui Zhang, Hazem Elmeleegy, Cecilia M. Procopiuc, and Divesh Srivastava. Reverse engineering complex join queries. In SIGMOD, pages 809–820, 2013.

[71] Moshé M. Zloof. Query by example. In AFIPS National Computer Conference, pages 431–438, 1975.