

Query From Examples: An Iterative, Data-Driven Approach to Query Construction

Hao Li¹ Chee-Yong Chan¹

¹Department of Computer Science
National University of Singapore

li.hao@nus.edu.sg,
chancy@comp.nus.edu.sg

David Maier²

²Department of Computer Science
Portland State University

maier@cs.pdx.edu

ABSTRACT

In this paper, we propose a new approach, called Query from Examples (QFE), to help non-expert database users construct SQL queries. Our approach, which is designed for users who might be unfamiliar with SQL, 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 a pair of inputs: a sample database D and an output table R which is the result of Q on D . As there will be many candidate queries that transform D to R , QFE winnows this collection by presenting the user with new database-result pairs that distinguish these candidates. Unlike previous approaches that use synthetic data for such pairs, QFE strives to make these distinguishing pairs as close to the original (D, R) pair as possible. By doing so, it seeks to minimize the effort needed by a user to determine if a new database-result pair is consistent with his or her desired query. We demonstrate the effectiveness and efficiency of our approach using real datasets from SQLShare, a cloud-based platform designed to help scientists utilize RDBMS technology for data analysis.

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 [11] provides a web-based interface to facilitate scientists posing SQL queries on their uploaded research data. However, writing SQL queries for such do-it-yourself data exploration remains a challenging task for non-expert database users; and this consideration has motivated several

recent research efforts to help users with query construction.

One approach 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 [10, 13]. For example, SDSS maintains a sample of popular user queries to facilitate query reuse, and SQLShare facilitates browsing and searching of SQL queries posted by users.

Another approach is to provide a query recommendation facility that can recommend entire queries based on a user's and other users' past queries recorded in a query log [6] or recommend query snippets for specified SQL clauses (e.g. tables in the from-clause, predicates in the where-clause) based on the partial query fragment that the user has typed and any past queries authored by users of the data [12, 17].

Both the query-browsing as well as query-recommender approaches require the users to be familiar with SQL, as they need to be able to read and write SQL queries. In addition, these approaches might not be applicable if the data being queried belongs to a private database that is used only by a single user.

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

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 desired output table of Q on the database D . As there will be multiple 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. 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. Olston et al. note the advantage of using realistic data in presenting examples of query execution to users [18]. We adopt a similar philosophy, trying to minimally modify the original data to generate new example pairs instead of generating synthetic data. The user can either provide D or use an existing dataset (such as the current database state). In the latter case, he or she might want to work with a subset, to keep

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/3.0/>. Obtain permission prior to any use beyond those covered by the license. Contact copyright holder by emailing info@vldb.org. Articles from this volume were invited to present their results at the 42nd International Conference on Very Large Data Bases, September 5th - September 9th 2016, New Delhi, India.

Proceedings of the VLDB Endowment, Vol. 8, No. 13
Copyright 2015 VLDB Endowment 2150-8097/15/09.

the result size manageable.

EXAMPLE 1.1. To illustrate our QFE approach, suppose that a user needs help to determine her target query Q for the following database-result pair (D, R) , where D consists of a single table.

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

Database D

For simplicity, assume that there is a set of three candidate queries, $QC = \{Q_1, Q_2, Q_3\}$, for Q , where each $Q_i = \pi_{name}(\sigma_{p_i}(Employee))$, with $p_1 = \text{'gender = "M"}$, $p_2 = \text{'salary > 4000'}$, and $p_3 = \text{'dept = "IT"}$. To help identify the user's target query among these three candidates, our approach will first present to the user a modified database¹ D_1 and two possible query results, R_1 and R_2 , on D_1 :

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

Database D₁

name
Darren

Result R₂

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 query in $\{Q_2\}$ producing the result R_2 . 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 another round and present the user with another modified database D_2 and two possible results, R_3 and R_4 on D_2 :

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

Database D₂

name
Darren

Result R₄

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 with at most two rounds of user feedback, each of which involves a single change in the database. \square

There are two main challenges for the QFE approach. The first is how to generate candidate target queries given an initial database-result pair; and the second is how to optimize the user-feedback interactions to minimize the user's effort to identify the desired query. In this paper, our focus is on the second challenge, as existing techniques [21, 23] are available to address the first.

Our paper makes two key contributions. First, we propose a novel approach, *Query From Examples*, to help users

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

construct queries. For users who are not familiar with SQL, 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 demonstrate the effectiveness and efficiency of our approach using real datasets from SQLShare [11], a cloud-based platform designed to help scientists utilize RDBMS technology for data analysis.

As a first step towards the QFE approach, our current implementation is limited to supporting only simple select-project-join (SPJ) queries, and we have not fully explored opportunities to improve execution-time performance. We plan to investigate these issues as part of our future work.

The rest of this paper is organized as follows. We present an overview of our new QFE approach in Section 2, and discuss its details in Sections 3 to 5. Section 6 presents additional extensions. An experimental evaluation of QFE is presented in Section 7. Section 8 discusses related work. Finally, we conclude in Section 9.

2. OUR APPROACH

Figure 1 illustrates the overall architecture of our approach. 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 $QC = \{Q_1, \dots, Q_n\}$ 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, 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, \dots, 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, \dots, 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. 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, 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 the entire new database D' and query results R_1, \dots, 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 1.

The overall procedure for QFE is shown in Algorithm 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 1), it means that the target query is not in the initial set of candidate queries QC . In this case, QFE will initiate another round of candidate-query

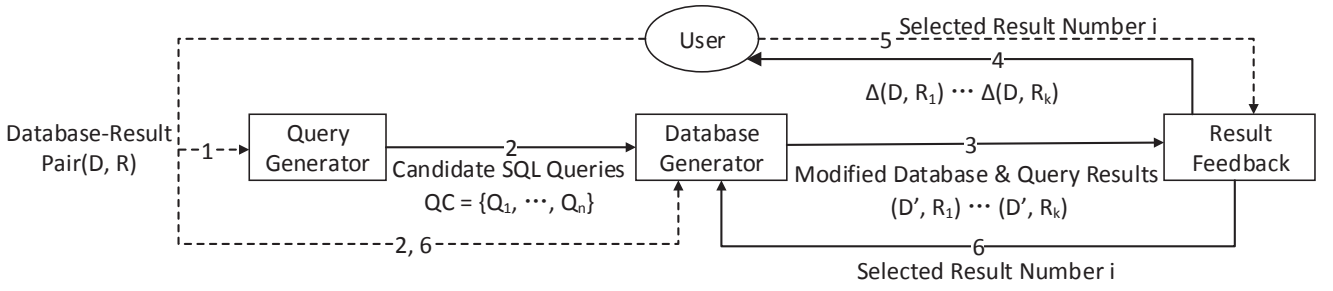


Figure 1: Overall Architecture of QFE

Algorithm 1: QFE

Input: A database-result pair (D, R)
Output: Target query

- 1 $QC = \text{Query-Generator}(D, R)$
- 2 **repeat**
- 3 $D' = \text{Database-Generator}(D, QC)$
- 4 $QC = QC_1 \cup \dots \cup QC_k$ // 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 $x = \text{Result-Feedback}(D', R_1, \dots, R_k)$
- 8 $QC = QC_x$
- 9 **until** $|QC| = 1$
- 10 **return** Q where $QC = \{Q\}$

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

For the 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, \dots, 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 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 D' , and then present the details of the key components of QFE.

3. 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 $\{QC_1, \dots, QC_k\}$, whose query results $\{R_1, \dots, R_k\}$ 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 $C = \{QC_1, \dots, QC_k\}$

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 $\{|QC_1|, \dots, |QC_k|\}$. 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:

$$cost(D') = currentCost + residualCost \quad (1)$$

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

$$currentCost = dbCost + resultCost \quad (2)$$

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

$$dbCost = minEdit(D, D') + \beta \times n \quad (3)$$

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

$$resultCost = \sum_{i=1}^k minEdit(R, R_i) \quad (4)$$

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

$$cost(D') = minEdit(D, D') + \beta \cdot n + \sum_{i=1}^k minEdit(R, R_i) + N \times (minEdit(D, D')/\mu + \beta) + \frac{2}{k} \sum_{i=1}^k minEdit(R, R_i) \quad (5)$$

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.

3.1 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

$$N = \log_2(\max\{|QC_1|, \dots, |QC_k|\}) \quad (6)$$

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. \square

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

$$N = N_1 + N_2 \quad (7)$$

$$N_1 = \lfloor (\max\{|QC_1|, \dots, |QC_k|\})/x \rfloor - 1 \quad (8)$$

$$N_2 = \lceil \log_2(\max\{|QC_1|, \dots, |QC_k|\}) - xN_1 \rceil \quad (9)$$

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 (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 (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 (6) for the estimation of N .

4. 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 [21, 23]. In this paper, we adopted the QBO approach of Tran et al. [21] 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. [23].

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 possible².

²In practice, it might be better to set these parameters conservatively, then relax them if more candidate queries are needed.

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 join³ 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., $p = p_1 \vee \dots \vee p_m$, where each p_i is a conjunction of one or more terms and a term is a comparison between an attribute and a constant.

5. 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, $QC = QC_1 \cup \dots \cup QC_k$, such that all the queries in each QC_i generate the same output result R_i on D' , and R_1, \dots, 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 T being the foreign-key join of all the relations in the database D . Thus, since R determines the projection list ℓ , all the queries in QC are essentially different selection queries on the single relation T . 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 database D' is valid (i.e., D' does not violate any integrity constraints). We discuss how to relax these assumptions in Section 6.

5.1 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 $T(A_1, \dots, A_n)$ and a set of queries QC . For each attribute A_i in T , 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 5.1. Consider a relation $T(A, B, C)$ where both A and B have numeric domains; and a set of queries $QC = \{Q_1, Q_2\}$, where $Q_1 = \sigma_{(A \leq 50) \wedge (B > 60)}(T)$ and $Q_2 = \sigma_{(A \in \{40, 80\}) \wedge (B \leq 20)}(T)$. 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 $\mathcal{P}_{QC}(C) = \{[-\infty, \infty]\}$. \square

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

EXAMPLE 5.2. Consider a relation $T(A, B, C)$ 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 $Q_1 = \sigma_{A \in \{b, c, e\}}(T)$ and $Q_2 = \sigma_{A \in \{a, b, d, e\}}$

³If 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., [16]).

(T). 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 : $\mathcal{P}_{QC}(A) = \{\{a, d\}, \{b, e\}, \{c\}, \{f, g\}\}$. \square

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

EXAMPLE 5.3. Continuing with Example 5.1, $TC = ((40, 50], [-\infty, 20], [-\infty, \infty])$ is an example of a tuple class for T , and $(48, 3, 25) \in TC$. \square

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 T 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 T by a pair of tuple classes (s, d) of T to represent that some tuple $t \in T$, 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 5.1. 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, $QC = QC_1 \cup QC_2 \cup \dots \cup QC_m$, $m \in [1, 4^n]$, 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. \square

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

Algorithm 2: Database-Generator

Input: A database D , a set of candidate queries QC

Output: A modified database D'

- 1 $SP = \text{Skyline-STC-DTC-Pairs}(D, QC)$
 - 2 $S_{opt} = \text{Pick-STC-DTC-Subset}(SP, QC)$
 - 3 Let D' be a modified database generated from D and S_{opt}
 - 4 **return** D'
-

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 $cost(S) = cost(D')$.

5.2 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, 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 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} .

5.3 Algorithm Skyline-STC-DTC-Pairs

The function **Skyline-STC-DTC-Pairs**, shown in Algorithm 3, takes the initial database D and a set of candidate queries QC as inputs to generate a set of skyline (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 T formed by joining all the relations in D based on

Algorithm 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 $SP = \emptyset$
 - 3 initialize $minbalance = \infty$
 - 4 let n be the number of distinct selection-predicate attributes in QC
 - 5 **for** $i = 1$ **to** n **do**
 - 6 | initialize $SP_i = \emptyset$
 - 7 | **foreach** $s \in STC$ **do**
 - 8 | | let $DTC =$ set of destination-tuple classes that can be derived from s by modifying i subsets
 - 9 | | **foreach** $d \in DTC$ **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 | | | | $SP_i = SP_i \cup \{p\}$
 - 16 | $SP = SP \cup SP_i$
 - 17 | **if** the running time is larger than threshold δ **then**
 - 18 | | **break**
 - 19 **return** SP
-

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 T 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 i^{th} 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 i^{th} 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.

5.4 Algorithm Pick-STC-DTC-Subset

The function **Pick-STC-DTC-Subset**, shown in Algorithm 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

Algorithm 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  $L = \emptyset$ 
2 initialize  $mincost = \infty$ 
3 foreach  $p \in SP$  do
4   if  $cost(\{p\}) < mincost$  then
5      $L = \{\{p\}\}$ 
6      $mincost = cost(\{p\})$ 
7   else if  $cost(\{p\}) == mincost$  then
8      $L = L \cup \{\{p\}\}$ 
9 initialize  $OP_1 = SP$ 
10 for  $i = 2$  to  $|SP|$  do
11   initialize  $OP_i = \emptyset$ 
12   foreach  $op \in OP_{i-1}$  do
13     foreach  $p \in SP, p \notin op$  do
14        $op' = op \cup \{p\}$ 
15       if  $balance(op') < balance(op)$  then
16          $OP_i = OP_i \cup \{op'\}$ 
17       if  $cost(op') < mincost$  then
18          $L = \{op'\}$ 
19          $mincost = cost(op')$ 
20       else if  $cost(op') == mincost$  then
21          $L = L \cup \{op'\}$ 
22 let  $S_{opt} \in L$  such that  $balance(S_{opt}) \leq balance(S) \forall S \in L$ 
23 return  $S_{opt}$ 
```

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 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.

5.4.1 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 (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 5.4. Consider the following joined relation $T = T_1(\underline{A}, B, C) \bowtie_A T_2(A, D)$, 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

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

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 T . \square

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 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 $cost(S)$. To minimize $resultCost$, tuple-class modifications that have no side-effects are preferred.

6. DISCUSSION

We first discuss in Sections 6.1 to 6.3 how our approach can be generalized by relaxing the three assumptions stated in Section 5. We conclude with a discussion of how our approach can be extended to support more expressive queries in Section 6.4.

6.1 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 QFE solution can handle set semantics by adopting the second approach. Further research is required to incorporate the first approach as well into QFE.

6.2 Queries with Different Join Schema

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 schema 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 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 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.

6.3 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.

6.4 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.

7. 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), and $\delta = 1s$ for the time threshold parameter in Algorithm 3. We examine the sensitivity of these parameters in Sections 7.3 and 7.4.

Sections 7.2 to 7.6 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 7.7 briefly reports additional experimental results.

7.1 Database and Queries

Our experiments were conducted using two real datasets. The first dataset is a scientific database of biology information taken from SQLShare⁴ 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 Baseball⁵. 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.

$$\begin{aligned}
Q_1 &= \pi_{*}(\sigma_{P.logFC_{Fe} < 0.5 \wedge P.logFC_{Fe} > -0.5 \wedge P.logFC_P < -1} \\
&\quad \wedge P.logFC_{Si} < -1 \wedge P.logFC_{Urea} < -1} \\
&\quad \wedge (P.PValue_{Fe} < 0.05 \vee P.PValue_P < 0.05 \vee P.PValue_{Si} < 0.05 \\
&\quad \vee P.PValue_{Urea} < 0.05)) \\
&\quad (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok) \\
Q_2 &= \pi_{*}(\sigma_{P.logFC_{Fe} < 1 \wedge P.logFC_P > 1 \wedge P.logFC_{Si} > 1} \\
&\quad \wedge P.logFC_{Urea} > 1 \wedge (P.PValue_{Fe} < 0.05 \vee P.PValue_P < 0.05 \\
&\quad \vee P.PValue_{Si} < 0.05 \vee P.PValue_{Urea} < 0.05)) \\
&\quad (PmTE_ALL_DE(P) \bowtie table_Psemu1FL_RT_spgp_gp_ok) \\
Q_3 &= \pi_{managerID, year, R}(\sigma_{teamID = "CIN" \wedge year > 1982 \wedge year \leq 1987} \\
&\quad (Manager \bowtie Team)) \\
Q_4 &= \pi_{ManagerID, year, 2B}(\sigma_{playerID = "sotoma01" \vee playerID = \\
&\quad "brownto05" \vee playerID = "pariske01" \vee playerID = "welshch01"} \\
&\quad (Manager \bowtie Team \bowtie Batting)) \\
Q_5 &= \pi_{ManagerID, year, HR}(\sigma_{playerID = "rosepe01" \wedge HR > 1 \wedge 2B < 3} \\
&\quad (Manager \bowtie Team \bowtie Batting)) \\
Q_6 &= \pi_{ManagerID, year, 3B}(\sigma_{playerID = "esaskni01" \wedge (IP > 4380 \\
&\quad \vee (IP < 4380 \wedge BBA < 485))} (Manager \bowtie Team \bowtie Batting))
\end{aligned}$$

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) .

7.2 Results for Default Settings

In this section, we present experimental results for the default settings with $\beta = 1$ and $\delta = 1s$, where the largest query subset is always chosen at each iteration. Due to space constraints, we discuss only the results for the scientific database; the results for the baseball database will be partially presented in Section 7.3.

Both Q_1 and Q_2 require 6 iterations of result feedback with our prototype. Table 1 shows the following per-round performance statistics: (1) the number of candidate queries

⁴<http://escience.washington.edu/sqlshare>

⁵<http://www.seanlahman.com/baseball-archive/statistics>

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
<i>dbCost</i>	1	2	2	1	2	8
<i>resultCost</i>	12	11	12	11	13	80
<i>avgResultCost</i>	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
<i>dbCost</i>	1	2	2	2	1	2
<i>resultCost</i>	11	9	10	11	11	12
<i>avgResultCost</i>	5.5	4.5	5	5.5	5.5	6

(b) Results for Query Q_2

Table 1: Per-round statistics for scientific database.

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; (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).

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.

7.3 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) 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 2. 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.

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 2: Effect of β for baseball database

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 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 .

7.4 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 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(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(b), the execution time for Q_2 decreases when δ increases from 0.1 to 0.2 for the same reason.

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 $\delta = 2s$.

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: Effect of δ for scientific database

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

7.5 Efficiency of Algorithm 4

In this section, we examine the efficiency of Algorithm 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 4 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 4 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 4: Performance of Algorithm 4 for scientific database

The results show that the running times of Algorithm 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 $\delta = 1$ s. 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 5 compares the execution timings of Algorithm 4 for these 5 subsets.

# of skyline pairs	200	400	600	800	1000
Exec. time (s)	3.22	24.55	65.76	104.54	156.49

Table 5: Execution time of Algorithm 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.

7.6 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, \dots, S_6) such that $S_1 \subset S_2 \subset \dots \subset S_6$ and $Q_2 \in S_1$. The cardinality of these query subsets and their performance results are shown in Table 6.

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 6: 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 7 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 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 7: Breakdown of first iteration’s running time (in sec)

Observe that the running time is dominated by the first and third steps, with Algorithm 4 incurring the least amount of time. The results demonstrate that our approach can scale for a reasonably large number of candidate queries.

7.7 Other Experiments

In this section, we briefly discuss the results of three additional experiments that we have conducted. The first two experiments aim to find out how sensitive our approach is to the user’s choice of the initial (D, R) pair, and the last experiment is a preliminary user study to evaluate the effectiveness of our approach. Due to space constraints, the details are given elsewhere [14].

Effect of size of initial database-result pair. This experiment evaluated the effect of the size of the initial database-result pair on performance. For a given initial database-result pair (D, R) , we created four subsets of D : D_1, D_2, D_3, D_4 ; where $D_4 = D$, and for $i \in [1, 3]$, we have

$|D_i| = \frac{i}{4} \times |D_4|$ and $Q(D_i) \subseteq Q(D_{i+1})$. We measured the performance using each of the initial database-result pair $(D_i, Q(D_i))$, $i \in [1, 4]$, in terms of the number of iterations, modification cost, and execution time. From our experimental results, we did not observe any clear trend in the effect of the size of initial database-result pair on performance. For some queries, using the largest initial database-result pair gave the best performance; while for other queries, the best performance was obtained for using the smallest initial database-result pair. In terms of the running time, the best-performance execution time could be up to 2.2 times faster than the worst-performance execution time.

Effect of entropy of attributes’ active domains. This experiment evaluated the effect of the entropy of an attribute’s active domain on performance. For an initial database-result (D, R) pair for a target query Q , we selected an attribute $T.A$ that appears as a selection attribute in many of the candidate queries for (D, R) , and created five datasets (denoted by D_1, \dots, D_5) which are equivalent except for the number of distinct values in $T.A$. Specifically, let T_i denote the instance of relation T in D_i , $i \in [1, 5]$. The datasets were created such that $\pi_A(T_1) = \pi_A(T)$, and $|\pi_A(T_i)| = \frac{6-i}{5} \times |\pi_A(T_1)|$, $i \in [1, 5]$. Thus, $\pi_A(T_i) \supset \pi_A(T_{i+1})$, $i \in [1, 5]$. Furthermore, $Q(D_i) = Q(D_j)$ for any $i, j \in [1, 5]$. Our experimental results show that there is no clear trend in the effect of the entropy of attributes’ active domain on performance. In terms of the running time, the best-performance execution time could be up to 1.3 times faster than the worst-performance execution time.

User Study. We also conducted a preliminary user study to evaluate the feasibility of our approach. This study involved three participants (all of whom were CS graduate students) and used three synthetic target queries over the Adult relation (containing 5227 tuples) extracted from the 1994 Census database⁶. This dataset was chosen over the scientific and baseball datasets as we felt that its data domain would be easier to understand for users. To evaluate the effectiveness of our cost-based approach, we compared it against an alternative cost model that aims to reduce both the size of query subsets as well as the number of iterations by choosing data modifications to maximize the number of partitioned query subsets. Thus, for each target query Q , each participant used two different approaches to determine Q ; the order of interaction with the two approaches were alternated for each query to ensure fairness.

All the participants succeeded in determining the target queries. The total time taken for each iteration was dominated by the user response time, which was on average 92.4% of the total time. The longest and shortest user response times were, respectively, 85 seconds and 2 seconds. The highest and lowest modification costs were, respectively, 5 and 3. The experimental results also demonstrated the effectiveness of our cost model: although using the alternative cost model enabled the target queries to be determined with fewer iterations, the total execution time incurred by the alternative approach was longer. Our proposed approach was up to 1.5 times faster than the alternative approach. Overall, our preliminary study demonstrates the feasibility of our approach as all the participants were able to effectively determine the target queries with reasonable effort. Moreover, the comparison against an alternative approach

also suggests that our cost-based approach is effective.

8. RELATED WORK

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), their familiarity with the database schema, the type of help provided (e.g., query recommendation, query completion), and the resources available to help with the query construction process (e.g., whether query logs of past queries are available).

Several approaches [2, 7, 9] make query recommendations based on queries from other users who have similar information needs, which are maintained in a query log. Another direction studied is query auto-completion [12, 17], which aims to interactively help users compose their queries. As users type an attribute or table name, the system will automatically provide several available query fragments, such as selection or join predicates, on the fly, based on the frequency of fragments or schema information. However, such solutions are based on the users’ previous actions, and not on the users’ query intention. Another approach that has been proposed is *query reuse systems* [10, 13]. The idea here is to store users’ previous queries in a shared repository so that those users (or new users) can later browse them when constructing new queries. Our QFE approach differs from all the above approaches in that it does not require users to be familiar with SQL and also does not rely on the availability of query logs.

The broad idea of an example-driven approach for problem solving has been applied in diverse contexts (e.g., [3, 4, 8, 19, 22]). Dimitriadou et al. proposed an interactive, example-driven approach named *AIDE* to help users explore their databases, which is related to the general framework for automatic navigation of databases first introduced in [5]. *AIDE* 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, *AIDE* presents the user with a sample of tuples for feedback on which of those 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 differs from *AIDE* in three key aspects. First, our context is different, 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 differs from *AIDE* as QFE operates by first generating a set of candidate queries and then pruning away false positives via user feedback on possible query results shown in each iteration. In addition, QFE also generates a modified database in each iteration to distinguish different subsets of candidate queries. In contrast, *AIDE* generates a plausible query (out of possibly many candidate queries) using classification techniques, and their focus is not on distinguishing away alternative candidate queries. Third, *AIDE* only supports queries on a single relation, whereas our approach is more general.

⁶<http://archive.ics.uci.edu/ml/datasets/Adult>

Example-driven techniques have also been applied for debugging scheme mappings [3, 4]. Users are shown examples to differentiate alternative mapping specifications, and the system finds the desired mapping based on the user’s interests of these data examples. Although we also show different query outputs to help the user pick the correct query from the candidates, the methods are different. Unlike schema mapping, we need to modify the database to distinguish the false positive SPJ queries. Qian et al. also proposed a system for sample-driven schema mapping [19]. 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 et al. have developed example-driven techniques to solve many diverse problems. For instance, they have applied example-driven techniques to reformat text documents [22]. Due to the different contexts, the techniques developed there are not applicable to our work.

There is also related work on generating database to distinguish queries. Mannila and R  ih   first introduced a method to distinguish one query Q from a set of queries \mathcal{Q} [15]. Related to the well-known concept of an *Armstrong database*, they defined the notion of a *complete test database* for the query Q , which is used to establish the non-equivalence of Q and Q_i , for every $Q_i \in \mathcal{Q}$. Their technique is not applicable to our work because the set of queries being distinguished are more restricted as each Q_i is formed from Q by removing some selection predicate(s), and the selection predicates must not be disjunctive.

Shah et al. addressed the problem of generating test data to distinguish an input query from a set of so-called mutant queries to help users learn SQL [20]. The mutant queries are pre-defined using certain query templates, such as join-outerjoin mutations (e.g., change equijoin to outer join), comparison operator mutations (e.g., change $<$ to \leq), and so on. Thus, the technique developed for this specific problem is not applicable to our work.

9. CONCLUSION

In this paper, we have developed a new approach, called Query from Examples (QFE), to help non-expert database users construct SQL queries. 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 , 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 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.

Acknowledgements We would like to thank the reviewers

for their constructive comments. This research is supported in part by NUS Grant R-252-000-512-112 and by a Shaw Foundation Visiting Professorship.

10. REFERENCES

- [1] Sloan digital sky survey. <http://www.sdss.org/>.
- [2] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman. SQL QueRIE recommendations. *PVLDB*, 3(1-2), 2010.
- [3] B. Alexe, L. Chiticariu, R. J. Miller, and W. C. Tan. Muse: Mapping understanding and design by example. In *ICDE*, 2008.
- [4] B. Alexe, L. Chiticariu, and W.-C. Tan. Spider: A schema mapping debugger. In *VLDB*, 2006.
- [5] U.   etintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.
- [6] G. Chatzopoulou, M. Eirinaki, and N. Polyzotis. Query recommendations for interactive database exploration. In *SSDBM*, 2009.
- [7] G. Chatzopoulou et al. The QueRIE system for personalized query recommendations. *IEEE Data Eng. Bull.*, 34(2), 2011.
- [8] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-example: An automatic query steering framework for interactive data exploration. In *SIGMOD*, 2014.
- [9] A. Giacometti, P. Marcel, E. Negre, and A. Soulet. Query recommendations for OLAP discovery driven analysis. In *DOLAP*, 2009.
- [10] B. Howe, G. Cole, N. Khoussainova, and L. Battle. Automatic starter queries for ad hoc databases. In *SIGMOD(demo)*, 2011.
- [11] B. Howe, G. Cole, E. Souroush, P. Koutris, A. Key, N. Khoussainova, and L. Battle. Database-as-a-service for long-tail science. In *SSDBM*, 2011.
- [12] N. Khoussainova et al. Snipsuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1), 2010.
- [13] N. Khoussainova, Y. Kwon, W.-T. Liao, M. Balazinska, W. Gatterbauer, and D. Suciu. Session-based browsing for more effective query reuse. In *SSDBM*, 2011.
- [14] H. Li, C.-Y. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. Technical report, National University of Singapore, August 2015. <http://www.comp.nus.edu.sg/~chancy/techreport-august-2015-qfe.pdf>.
- [15] H. Mannila and K.-J. R  ih  . Automatic generation of test data for relational queries. *J. Comput. Syst. Sci.*, 38(2), 1989.
- [16] F. D. Marchi, S. Lopes, and J.-M. Petit. Efficient algorithms for mining inclusion dependencies. In *EDBT*, 2002.
- [17] A. Nandi and H. V. Jagadish. Assisted querying using instant-response interfaces. In *SIGMOD*, 2007.
- [18] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *SIGMOD*, 2008.
- [19] L. Qian, M. J. Cafarella, and H. V. Jagadish. Sample-driven schema mapping. In *SIGMOD*, 2012.
- [20] S. Shah et al. Generating test data for killing SQL mutants: A constraint-based approach. In *ICDE*, 2011.
- [21] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query reverse engineering. *The VLDB Journal*, 23(5), 2014.
- [22] K. Yessenov, S. Tulsiani, A. Menon, R. C. Miller, S. Gulwani, B. Lampson, and A. Kalai. A colorful approach to text processing by example. In *UIST*, 2013.
- [23] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.