CDMTCS
Research
Report
Series

LECQTER: Learning Conjunctive SQL Queries Through Exemplars

Stefan Böttcher
University of Paderborn
Paderborn, Germany

Sebastian Link
University of Auckland,
Auckland, New Zealand

Lin Zhang
University of Auckland,
Auckland, New Zealand

CDMTCS-453
February 2014

Centre for Discrete Mathematics and
Theoretical Computer Science

LECQTER: Learning Conjunctive SQL Queries Through Exemplars

Stefan Böttcher
University of Paderborn, Germany
stb@uni-paderborn.de

Sebastian Link
The University of Auckland, Private Bag 92019, New Zealand
s.link@auckland.ac.nz

Lin Zhang
The University of Auckland, Private Bag 92019, New Zealand
l.zhang@auckland.ac.nz

February 12, 2014

Abstract

We present Lecqter, a system for learning how to write sound conjunctive SQL queries by example. The key novelty of Lecqter is its ability to construct for every given conjunctive query Q without self-joins over every given database schema, a database db_Q such that for every query Q' in a large fragment of conjunctive queries, Q and Q' produce matching answers on every database if and only if Q and Q' produce matching answers on db_Q. Since it is known that such construction is impossible to achieve under set semantics, the key novelty relies on the use of SQL's bag semantics. Lecqter shows the answer to both the user query Q' and the target query Q, such that users receive immediate feedback that either their query is correct - and not just their query answer - or where their query answer deviates from that of the target query. Lecqter can therefore automate feedback and assessment in its primary application area of Massive Open Online Courses. Everyone who requires basic SQL skills to match the demands of our data-centric society can use Lecqter to confidently learn how to write sound conjunctive queries under the semantics of the industry standard.

Keywords: Armstrong database, Bag Semantics, Complexity, Conjunctive Query, Example Database, Learning, Set Semantics, SQL, Performance

1 Introduction

Data has evolved into becoming the number one asset for many organizations. The ability to write sound SQL queries is no longer a craft reserved for specialists, but a necessary core skill that businesses seek from more and more of their employees. Similarly, an increasing number of people working in education, engineering, entertainment, finance, government, health, science and so on must be able to query data to help achieve their objectives. The reliable relational technology that has been readily accessible through the industry standard SQL for more than three decades now is predicted to also dominate the market over the next years: According to a forecast by Gartner, the NoSQL market is estimated to be worth 3.5 billion US dollars and the SQL market is estimated to reach 40 billion US dollars annually by 2018 [5]. As Sean Doherty, Vice President at EnterpriseDB, puts it: “Relational databases may not be hot or sexy but for your important data, there is no substitute” [5].

The are many forms in which basic SQL skills can be acquired: certified training courses, self- and peer-learning, or in a so-called Massive Open Online Course (MOOC), for examples. Independently of the learning environment, the quality, timing and form of feedback is critical to effective learning. Intuitively, the larger the number of trainees the more feedback relies on self- or peer-review. This form of feedback, however, is of low quality and its use for assessment is therefore difficult to justify. *Automation* is seen as the savior of high quality feedback and assessment, in particular in MOOCs [12].

There are strong economic reasons to pursue research in automating feedback and assessment now. As Moshe Vardi wrote “From the point of view of Silicon Valley, higher education is a particularly fat target right now. MOOCs may be the battering ram of this attack” [22].According to the National Venture Capital Association, investment in education technology companies increased from less than \$100 million in 2007 to nearly \$400 million last year [1].

Automated feedback and assessment must overcome several obstacles, in particular for learning how to write sound SQL queries. However, the benefits of pushing the boundaries further provide great motivation. Part of its strength as a query language is derived from providing users with different ways of declaring their target in SQL syntax. The trade-off for trainers is therefore the need to verify the soundness of many different sound queries and to provide different feedback to different incorrect queries. The savings in resources obtained from higher levels of automation thus provide strong economic incentives, in particular for MOOCs [12]. Currently, however, most systems simply verify whether the answer to a user query is the same as that of the target query on a given database. Obviously, if the answers deviate from one another, the user query is incorrect. Unfortunately, if the answers are the same, neither trainers nor trainees can usually conclude that the user query is semantically correct, in the sense that it is equivalent to the target query, that is, returns matching answers on every database. It follows that the current level of feedback, provided by SQL tutoring systems, is of poor quality. Ideally, the database, on which the user queries are evaluated, could be chosen such that users would indeed be able to conclude that their query is semantically correct whenever it returns answers that match those of the target query. In fact, such a database would be an ideal model of the target query. For this reason, we call such a database an *exemplar*. The idea of exemplars faces several challenges that seem to inhibit their practical use to provide high quality assessment and feedback for learning how to write sound SQL queries. Firstly, neither can real-world data be expected to form exemplars - as illustrated later - nor is the complexity of recognizing exemplars known. Secondly, Mannila and Räihä have given an example of a simple conjunctive target query Q without self-joins and two conjunctive user queries Q_1 and Q_2, neither of them equivalent to Q, for which no exemplar exists [17]. That is, no matter which database is chosen, Q_1 returns the same answer as Q or Q_2 returns the same answer as Q. Interestingly, Mannila and Räihä have assumed set semantics where duplicate tuples are always removed from the results. The default of the industry standard SQL, however, is the use of bag semantics, where duplicates are preserved. The main drivers for this default semantics are the significance of duplicates for aggregate queries, and the cost of duplicate removal. It is the key observation of this research that the use of bag semantics guarantees the existence of exemplars for conjunctive queries without self-joins. Our contributions, as outlined now, establish exemplars as a useful tool to provide high-quality feedback and assessment for learning how to write semantically sound conjunctive SQL queries.

- We revive the idea of exemplars as a core concept to automate feedback and assessment for learning how to write sound SQL queries. We believe that exemplars provide further practical motivation to study deep questions in core database research, and to establish links to database education, which is a big target area of the software industry.

- Mannila and Räihä investigated the existence of exemplars for conjunctive queries Q without self-joins. More specifically, they focused on the language $\mathcal{L}(Q)$ that consists of all conjunctive queries that use the same `SELECT` and `FROM` clause as those of Q, respectively, and where every constant that appears in the `WHERE` clause of Q' is also a constant in the `WHERE` clause of Q. Indeed, they investigated the existence of an $\mathcal{L}(Q)$-exemplar db_Q in the sense that for each query $Q' \in \mathcal{L}(Q)$, the answer to Q' when evaluated on db_Q is different from the answer to Q when evaluated on db_Q. It was shown that, under set semantics, there are conjunctive queries Q without self-joins and $Q_1, Q_2 \in \mathcal{L}(Q)$ for which no $\mathcal{L}(Q)$-exemplars exist [17]. As our main result we show, under bag semantics, how to construct an $\mathcal{L}(Q)$-exemplar for every conjunctive query without self-joins.

- We have fully implemented our construction in form of a graphical user interface (GUI), called Lecqter. In a showcase we illustrate the primary use of Lecqter as an automated tutoring system. Trainers can enter SQL table definitions, followed by a target query in the GUI. Lecqter then generates an exemplar over the table definition within a MySQL database. Trainees can then enter their queries in the GUI, which connects to the MySQL database to fetch the query answer and present it in comparison to the target answer. As the distinguishing key feature of Lecqter, its users recognize the semantic soundness of their queries whenever the query answers on the exemplar match those shown for the target query. Otherwise, the difference in the query answers is highlighted visually.

- We demonstrate the efficiency of constructing our $\mathcal{L}(Q)$-exemplars for the purpose of learning how to write sound conjunctive SQL queries in theory and by a series of experiments. These are established despite challenging problems well-known from the literature, as outlined in Section 3. While the worst-case time and space complexity of our construction is exponential in the total number of participating attributes, the exemplar separates a number of pairwise non-equivalent user queries from the target query that is at least exponential in the exemplar's size. Our experiments show that exemplars were constructed within 25 seconds for 15 attributes in these worst cases. On average, exemplars were constructed within 350ms for 15 attributes. It is a feature of our construction that its size and time are both proportional to the number of conditions in the `WHERE` clause of the target query. Therefore, the number of conditions is a natural mechanism to control the complexity of our construction. Our experiments show further that user queries can be efficiently evaluated on the constructed exemplars, taking 70ms to evaluate in the worst cases observed.

- We propose an alternative use of $\mathcal{L}(Q)$-exemplars for the training of conjunctive queries under set semantics, motivated by the `SELECT DISTINCT` clause in SQL. Indeed, while it is no longer possible to use the same database db_Q to distinguish Q simultaneously from all user queries in $\mathcal{L}(Q)$, for each $Q' \in \mathcal{L}(Q)$ we can select a single tuple from each table of db_Q that results in a database which separates Q from Q' if and only if they are not equivalent. Therefore, exemplars are also instrumental in automating high-quality feedback and assessment of conjunctive queries under set semantics.

In summary, Lecqter is an effective tool to efficiently provide high-quality feedback and assessment to users who want to learn how to write sound conjunctive SQL queries.
Organization. Motivating examples for our research are presented in Section 2. The related work discussed in Section 3 describes the current state-of-the-art and research challenges. Section 4 introduces preliminary concepts required by the discussion. The construction of exemplars is detailed in Section 5, and a showcase of Lecqter is given in Section 6. Experimental results are presented in Section 7, and the use of exemplars for set semantics is discussed in Section 8. Section 9 concludes, and future work is outlined in Section 10.

2 Motivating Examples

In this section three different motivations are provided for the need and use of Lecqter. Firstly, it is illustrated that current systems for learning how to write sound SQL queries do not provide high-quality feedback to trainees. This motivates research on automated SQL tutoring systems that do provide high-quality feedback. Secondly, a simple example illustrates convincingly that real-world data sets cannot be expected to separate a reasonably large number of non-equivalent queries. This motivates the use of Lecqter for constructing exemplars. In fact, exemplars can be seen as templates whose values can be replaced by real-world values. Thirdly, a natural example shows that the use of set semantics cannot even always produce exemplars that can separate three non-equivalent queries.

2.1 SQL Tutor

GNU SQLtutor is a web-based interactive tutorial of SQL. Trainees can select a tutorial, which presents a series of tutorial questions in a simple dialog. When finished SQL-tutor displays a final evaluation with the review of all questions asked during the session together with the user's SQL queries and correct answers for wrong solutions [7]. For instance, in the tutorial **Olympics** the trainees are asked to convert the following English language query into SQL syntax: "For the discipline table tennis in the singles Women event, show in which city 'Chen, Jing' won medals of which color". A semantically correct conjunctive SQL query is [7]:

```
SELECT city, medal
FROM   olympics
       JOIN medals USING (year)
       JOIN sports USING (discipline\_id, event\_id, category)
       JOIN athletes USING (athlete\_id)
WHERE  discipline = 'Table Tennis' AND event='singles Women' AND
       forename='Jing' AND surname='Chen';
```

which returns the correct answer [7]:

Seoul	gold
Atlanta	silver
Sydney	bronz

.

The semantically incorrect conjunctive SQL query

```
SELECT city, medal
FROM   olympics
       JOIN medals USING (year)
       JOIN sports USING (discipline\_id, event\_id, category)
       JOIN athletes USING (athlete\_id)
WHERE  event='singles Women' AND surname='Chen';
```

returns the same answer. Trainees who enter this query cannot conclude that their query is semantically correct. The reality is that most trainees will think that they submitted a semantically correct query. Worse, even trainees who submit indeed a semantically correct query cannot conclude that they were right. Consequently, the value for trainees in using such an automated tutor is modest at best. Similarly, the value for trainers in using such an automated tutor is low: automated feedback is minimal and automated marking difficult to justify.

2.2 Kinship Data Set

The **Kinship** data set from the UCI Machine Learning repository [2] consists of people with 24 unique names that belong to two families with two equivalent structures. The English language query "Print the names of Marco's daughters and their aunts that are Marco's sisters" can be written as the conjunctive SQL query:

```
SELECT s.pname1 AS 'aunt', d.pname1 AS 'daughter'
FROM   father f, sister s, daughter d
WHERE  f.pname1='Marco' AND f.pname1=d.pname2 AND
       s.pname2=d.pname2 AND d.pname1=f.pname2;
```

The non-equivalent query "Print the names of Marco's daughters and their aunts", which also includes the aunts that are sisters of Marco's wife, can be written as follows:

```
SELECT s.pname1 AS 'aunt', d.pname1 AS 'daughter'
FROM  father f, sister s, daughter d
WHERE f.pname1='Marco' AND s.pname2=d.pname2 AND d.pname1=f.pname2;
```

However, for the simple reason that Marco's wife does not have any sisters, both queries return the same result

aunt	*daughter*
Angela	Sophia

.

The example illustrates convincingly that real-life data cannot be expected to exhibit a structure able to separate a reasonably large number of non-equivalent queries.

2.3 Limits of Set Semantics

Finally, we adopt the example from [17, Example 14, page 250] to a real-world setting. Indeed, there cannot be any database db_Q such that the query Q:

```
SELECT DISTINCT name
FROM     PATIENT p, DIAGNOSIS d
WHERE    name='Smith' AND p.pid='NHI003' AND
         d.pid='NHI003' AND condition='HIV';
```

produces answers different from those of the queries Q_1 and Q_2 on db_Q. Here, Q_1 results from Q by removing from its `WHERE` clause `condition='HIV'`, and Q_2 results from Q by removing from its `WHERE` clause `name='Smith'`. Indeed, for $Q(db) \neq Q_1(db)$ to hold, the table over `PATIENT` must contain (Smith, NHI003), and the table over `DIAGNOSIS` must not contain (NHI003, HIV) but some (NHI003, c) where $c \neq$ `HIV`. For $Q(db) \neq Q_2(db)$ to hold, the table over `DIAGNOSIS` must contain (NHI003, HIV). Hence, there is no database db such that $Q(db) \neq Q_1(db)$ and $Q(db) \neq Q_2(db)$ both hold.

Summary. Together, the examples illustrate the core contribution of our research in relation to the current state-of-the-art: We show how the bag semantics of SQL enables LECQTER to construct databases that can separate a large number of non-equivalent queries from any given conjunctive query. Therefore, LECQTER can automate high-quality feedback to trainees and automate their assessment.

3 Related Work

Naturally, there is quite a number of SQL tutoring systems available, already on the Web alone. Systems such as SQLtutor [7], SQL exercises [19] and SQLZOO [18] are all interactive tutorials that provide a wide range of helpful exercises for SQL trainees. The main shortcoming is their inability to provide feedback on the semantic soundness of the queries submitted by their users. A positive exception to this list is the automated lab tutor GRADIENCE from Stanford [21], which includes on-line SQL exercises that "feature immediate constructive feedback about the correctness of the submitted queries". However, the detection of semantically incorrect queries depends crucially on the ability of the trainer to anticipate all potential cases. Acknowledging the difficulty of this problem, GRADIENCE offers some guidelines to overcome the "critical issue in the design of labs ... to pick the sample and evaluation databases. An important point is that students will tend to try to write queries that work on the sample database, even if they don't work in general. Thus, it is important to use an evaluation database that detects such attempts" [21]. The use of exemplars overcomes this design issue, as the sample database that is automatically constructed by LECQTER is ideal in the sense that it will detect any such attempts. Our recommendation is to automatically generate exemplars whenever they exist and their construction requires reasonable resources, and to use GRADIENCE in other cases. We hope that this article initiates future research that uncovers fragments of query languages for which exemplars can be constructed (efficiently).

Mannila and Räihä have initiated research on exemplars in [16, 17], where they were called *complete test databases*. The notion was characterized using *Armstrong databases* [6], and two constructions for producing complete test databases were given. The research in these articles was focused on set semantics only, and essentially the example from Section 2.3 was given to show that even $\{Q_1, Q_2\}$-complete test databases do not exist for some conjunctive queries without self-joins. That is, for every database, the answer to Q_1 coincides with that to Q or the answer to Q_2 coincides with that to Q. In sharp contrast, we show that $\mathcal{L}(Q)$-complete test databases do exist for every conjunctive query without self-joins under bag semantics. Note that the default bag semantics of SQL was not considered in [17], and neither was the possibility of using complete test databases as a basis to extract different databases for different user queries in order to separate them from the target query, see Section 8. Furthermore, our construction of exemplars is original.

The availability of an $\mathcal{L}(Q)$-exemplar db_Q means that for every $Q' \in \mathcal{L}(Q)$, Q and Q' are semantically equivalent if and only if the answer to Q on db_Q is the same as the answer to Q' on db_Q. Therefore, the construction of exemplars is intrinsically linked to the well-studied problem of query equivalence. Under set semantics the problem is undecidable for relational calculus queries, and NP-complete for conjunctive queries [3]. Under bag semantics the query equivalence problem has the same complexity as the graph isomorphism problem [4], which is one of the few problems for which it is still unknown whether it is in *P*, or *NP*-complete, or neither. The complexity of the containment problem for conjunctive queries under bag semantics remains still open after twenty years of dedicated research [4, 9, 10, 13, 14]. Our results on the efficiency of our construction should be viewed under the plethora of these computationally hard problems.

4 Preliminaries

In this section we fix preliminary definitions that are required for the remainder of this article. These include the relational model, conjunctive queries, and notions of containment and equivalence, all under bag and set semantics.

Bags. We assume a countably infinite set $\mathfrak{A}$ of elements that we call *attributes*. Each attribute $A \in \mathfrak{A}$ is associated with a *domain* $dom(A)$ that consists of countably many elements. The elements of the domain $dom(A)$ model the potential values that can occur in the column represented by A. A *relation schema* is a finite sequence $(A_1, \dots, A_n)$ of attributes together with a name R, denoted by $R(A_1, \dots, A_n)$. A *tuple* over $R(A_1, \dots, A_n)$ is an element of the Cartesian product over the associated domains, that is, an element of $dom(R) := dom(A_1) \times \cdots \times dom(A_n)$. A *bag* or *multiset* over $R(A_1, \dots, A_n)$, usually denoted by B, is a collection of tuples over $R(A_1, \dots, A_n)$ which can occur one or many times in the collection. For a bag B and tuple t over $R(A_1, \dots, A_n)$ we denote by $|t|_B$ the number of occurrences of t in B. For two bags B, B' over $R(A_1, \dots, A_n)$ the *bag inclusion* $B \subseteq_b B'$ holds if and only if for all tuples t over $R(A_1, \dots, A_n)$, $|t|_B \leq |t|_{B'}$. B and B' are equal, denoted by $B =_b B'$ if and only if $B \subseteq_b B'$ and $B' \subseteq_b B$ both hold. The *proper bag inclusion* $B \subsetneq_b B'$ holds if and only if $B \subseteq_b B'$ holds and there is some tuple t over $R(A_1, \dots, A_n)$ such that $|t|_B < |t|_{B'}$. For a bag B over $R(A_1, \dots, A_n)$ and attributes $A_{i_1}, \dots, A_{i_m} \in \{A_1, \dots, A_n\}$ we denote by $\pi^b_{A_{i_1},\dots,A_{i_m}}$ the *duplicate-preserving projection* of B onto $A_{i_1}, \dots, A_{i_m}$, and by $\pi_{A_{i_1},\dots,A_{i_m}}$ the *duplicate-eliminating projection* of B onto $A_{i_1}, \dots, A_{i_m}$. A *database schema* $\mathcal{D}$ is a finite set of relation schemata, and a *database* db over $\mathcal{D}$ assigns to each relation schema in $\mathcal{D}$ a finite bag over the relation schema.

Conjunctive SQL Queries. A *conjunctive SQL query* Q (without self-joins) over a database schema $\mathcal{D}$ is a statement of the form

$$\begin{array}{l} \texttt{SELECT } A_{i_1}, \dots, A_{i_m} \\ \texttt{FROM } R_1, \dots, R_k \\ \texttt{WHERE } C_1 \texttt{ AND } \dots \texttt{ AND } C_l \end{array} .$$

Here, we have relation schemata $R_i(A^i_1, \dots, A^i_{n_i}) \in \mathcal{D}$ for $i = 1, \dots, k$. We assume that $R_1, \dots, R_k$ are distinct. From here on, we denote the schema over the distinct attributes

$$R_1.A^1_1, \dots, R_1.A^1_{n_1}, \dots, R_k.A^k_1, \dots, R_k.A^k_{n_k}$$

as $R(A_1, \dots, A_n)$ where $n = \sum_{i=1}^k n_i$. In a conjunctive SQL query, we further have for each $j = 1, \dots, m$, $A_{i_j} \in \{A_1, \dots, A_n\}$, and for $h = 1, \dots, l$, C_h denotes either an attribute-attribute equality $A_r = A_s$ or an attribute-constant equality $A_r = c$ for some $A_r, A_s \in \{A_1, \dots, A_n\}$ and some $c \in dom(A_r)$. A tuple t over $R(A_1, \dots, A_n)$ *satisfies* the `WHERE` clause, denoted by $\models_t \varphi_Q$, if and only for $h = 1, \dots, l$, $t(A_r) = t(A_s)$, if $C_h := A_r = A_s$, or $t(A_r) = c$, if $C_h := A_r = c$. Here, $(A_{i_1}, \dots, A_{i_m})$ is the *answer schema* of Q.

For a database db over $\mathcal{D}$, where $r_1, \dots, r_k$ denote the bags over $R_1, \dots, R_k$, respectively, the *answer to the conjunctive SQL query Q on db* is defined as

$$Q(db) := \pi^b_{A_{i_1},\dots,A_{i_m}}(\{t \in r_1 \times \cdots \times r_k \mid \models_t \varphi_Q\}),$$

using duplicate-preserving projection π^b. As usual $r_1 \times \cdots \times r_k$ is the bag that is the (bag) *cross product* of $r_1, \ldots, r_k$.

A *conjunctive query* over $\mathcal{D}$ is defined as a conjunctive SQL query, except that we use `SELECT DISTINCT` instead of `SELECT`. Similarly, the *answer to the conjunctive query Q on db* with relations $r_1, \ldots, r_k$ is defined as

$$Q(db) := \pi_{A_{i_1},\ldots,A_{i_m}}(\{t \in r_1 \times \cdots \times r_k \mid \models_t \varphi_Q\}),$$

using duplicate-eliminating projection π.

Containment and Equivalence. For a database db over $\mathcal{D}$, a conjunctive SQL query Q over $\mathcal{D}$ with answer schema $(A_{i_1}, \ldots, A_{i_m})$, and a tuple t over $(A_{i_1}, \ldots, A_{i_m})$, let $|t|^Q_{db}$ denote the multiplicity of t in the answer $Q(db)$ to Q on db, i.e., the number of tuples in the bag $Q(db)$ that match t on all attributes of the output schema.

For two conjunctive SQL queries Q and Q' over database schema $\mathcal{D}$ and same answer schema $(A_{i_1}, \ldots, A_{i_m})$, Q is said to be *contained in* Q' (under bag semantics), denoted by $Q \subseteq_b Q'$, if and only if for every database db over $\mathcal{D}$, and every tuple $t \in dom(A_{i_1}) \times \cdots \times dom(A_{i_m})$, $|t|^Q_{db} \leq |t|^{Q'}_{db}$. Q is said to be *properly contained in* Q' (under bag semantics), denoted by $Q \subsetneq_b Q'$, if and only $Q \subseteq_b Q'$ holds, but $Q' \subseteq_b Q$ does not hold. Q and Q' are said to be *equivalent* (under bag semantics), denoted by $Q \equiv_b Q'$, if and only if $Q \subseteq_b Q'$ and $Q' \subseteq_b Q$ both hold. Let $\mathcal{L}$ denote a class of conjunctive queries. The *containment (equivalence) problem for* $\mathcal{L}$ is to decide for arbitrarily given $Q, Q' \in \mathcal{L}$, whether $Q \subseteq_b Q'$ ($Q \equiv_b Q'$) holds. Similar notions can easily be defined for conjunctive queries (under set semantics).

5 Construction of Exemplars

This section develops the main foundation of Lecqter. After giving the definition of $\mathcal{L}(Q)$-exemplars for conjunctive SQL queries, the requirements on the language $\mathcal{L}(Q)$ are discussed. In what follows we establish a theory for constructing a special class of $\mathcal{L}(Q)$-exemplars that we call canonical. Subsequently, the computation of canonical $\mathcal{L}(Q)$-exemplars is detailed, and the complexity of the computation discussed.

5.1 Exemplars and Assumptions

We start with the main definition of this paper.

Definition 1 *Let Q denote a conjunctive SQL query and db_Q a database, both over database schema $\mathcal{D}$. Let $\mathcal{L}$ denote a class of conjunctive SQL queries over $\mathcal{D}$. We say that db_Q is an $\mathcal{L}$-*exemplar *for Q if and only if for every $Q' \in \mathcal{L}$, $Q(db_Q) = Q'(db_Q)$ if and only if $Q \equiv_b Q'$.*

Therefore, trainees who want to verify whether their query $Q' \in \mathcal{L}$ is equivalent to the target query Q just need to check that Q' produces the same answer as Q on an $\mathcal{L}$-exemplar for Q. Indeed, the answers to Q and Q' coincide on *every* database if and only if the answers to Q and Q' coincide just on the exemplar. Hence, the exemplar as indeed an ideal model for testing equivalence between Q and any $Q' \in \mathcal{L}$.

The existence of exemplars depends critically on the choice of $\mathcal{L}$. Here, we focus on the same language $\mathcal{L}$ as in previous research [16, 17]. That is, for a conjunctive SQL query Q, $\mathcal{L} := \mathcal{L}(Q)$ consists of those conjunctive SQL queries Q' which have identical `SELECT` and `FROM` clauses as those of Q, respectively, and in which every constant that appears in an equality of the `WHERE` clause in Q' also appears as a constant in some equality of the `WHERE` clause in Q. As explained in [16] these restrictions are reasonable: Users are unlikely to introduce incorrect attributes in the `SELECT` clause or incorrect table schemata in the `FROM` clause. Furthermore, users are unlikely to introduce in their queries constants which are not mentioned in the natural language description of the target query Q [16]. In sharp contrast to [16, 17] we consider indeed `SELECT` instead of `SELECT DISTINCT` statements. In fact, `SELECT DISTINCT` statements prevent the general existence of $\mathcal{L}(Q)$-exemplars, as shown in [16, 17] and illustrated in Section 2.

5.2 Canonical Exemplars

Our ultimate aim is to compute an $\mathcal{L}(Q)$-exemplar for every conjunctive SQL query Q without self-joins. For this purpose our goal is to first understand what tuples can form the elements of such an exemplar.

Containers

The approach towards achieving this goal is embodied in the following definition of minimal proper containers.

Definition 2 *Let Q be a conjunctive query without self-joins, and let $Q' \in \mathcal{L}(Q)$. We call Q' a* container *of Q if and only if $Q \subseteq_b Q'$ holds. We call Q' a* proper container *of Q if and only if $Q \subsetneq_b Q'$ holds. A proper container Q' of Q is* minimal *if and only if for every proper container $Q'' \in \mathcal{L}(Q)$ of Q, $Q'' \subseteq_b Q'$ implies $Q'' \equiv_b Q'$. We denote by $\mathcal{M}(Q)$ the set of all minimal proper containers of Q.*

In the following we denote by $\mathcal{M}(Q)$ a set of queries that contains one representative from each equivalence class of $\mathcal{M}(Q)/\equiv_b$. This notation simplifies discussion and is sufficient for our construction. In particular, $\mathcal{M}(Q)$ is unique up to the equivalence of queries.

Example 1 *Consider an application domain where purchases of products by customers are recorded. Products have an identifier* `pid` *and a name* `pname`*, customers have an identifier* `cid` *and a name* `cname`*. Customers* `cid` *purchase products* `pid` *on a* `date` *at a* `cost`*. The schema $\mathcal{D}$ is thus given by* `CUSTOMER(cid,cname)`*,* `PRODUCT(pid,pname)`*, and* `BUY(cid,pid,date,cost)`*. The target query Q is to "Print the customer name, product name and cost of all purchases made by the customer with identifier* `c7` *on* `3/9/14`*", which can be written in SQL as follows:*

```
SELECT cname, pname, cost
FROM   CUSTOMER c, PRODUCT p, BUY b
WHERE  c.cid=b.cid AND c.cid='c7' AND
       b.pid=p.pid AND b.date='3/9/14';
```

The queries Q' on the left, and Q'' on the right:

```
SELECT cname, pname, cost          SELECT cname, pname, cost
FROM   CUSTOMER c,                 FROM   CUSTOMER c,
       PRODUCT p,                         PRODUCT p,
       BUY b                              BUY b
WHERE  b.cid=c.cid AND             WHERE  b.cid=c.cid AND
       b.pid=p.pid AND                    b.pid=p.pid;
       b.date='3/9/14';
```

are both proper containers of Q, but only Q' is minimal.

$Q' \in \mathcal{L}(Q)$ is equivalent to Q iff it is a container of Q but not a container of any of Q's minimal proper containers Q_i.

Lemma 1 *Let Q be a conjunctive query without self-joins, and let $Q' \in \mathcal{L}(Q)$. Then, $Q \equiv_b Q'$ if and only if $Q \subseteq_b Q'$ and for every $Q_i \in \mathcal{M}(Q)$, $Q_i \not\subseteq_b Q'$.*

Proof If $Q \equiv_b Q'$, then $Q \subseteq_b Q'$ and $Q' \subseteq_b Q$. If there was some minimal proper container Q_i of Q for which $Q_i \subseteq_b Q'$, then $Q' \subseteq_b Q \subseteq_b Q_i \subseteq_b Q'$. Hence, $Q \equiv_b Q_i$ which would contradict the definition of a proper container.

Assume that $Q \not\equiv_b Q'$ and $Q \subseteq_b Q'$. We need to show that there is some $Q_i \in \mathcal{M}(Q)$ such that $Q_i \subseteq_b Q'$ holds. From $Q \not\equiv_b Q'$ and $Q \subseteq_b Q'$ follows that $Q' \not\subseteq_b Q$, hence Q' is a proper container of Q. If $Q' \in \mathcal{M}(Q)$, then we are done. Otherwise Q' is not a minimal proper container. Therefore, there is some proper container Q'' of Q such that $Q'' \subseteq_b Q'$ and $Q'' \not\equiv_b Q'$. As every chain of proper containers is finite, Q' contains some minimal proper container of Q.

Duplicate-preserving Projection

The next result states an observation that holds the key to the construction of exemplars for conjunctive SQL queries.

Lemma 2 *Let B, B' denote two bags over $R(A_1, \ldots, A_n)$, such that $B \subsetneq_b B'$. Then $\pi^b_{A_{i_1},\ldots,A_{i_m}}(B) \subsetneq_b \pi^b_{A_{i_1},\ldots,A_{i_m}}(B')$.*

Proof This follows immediately from the definition of the proper bag inclusion $\subsetneq_b$, and the fact that the duplicate-preserving projection $\pi^b_{A_{i_1},\ldots,A_{i_m}}$ does not reduce the cardinality of the underlying bags.

Lemma 2 does not apply to duplicate-eliminating projections. A simple example over $R(A)$ are the bags $B = \{0\}$ and $B' = \{0, 0\}$, i.e., $B \subsetneq_b B'$ holds. As $\pi^b_\emptyset(B) = \{()\} \subsetneq_b \{(), ()\} = \pi^b_\emptyset(B')$ holds, we have $\pi_\emptyset(B) = \{()\} = \{()\} = \pi_\emptyset(B')$ under duplicate-eliminating projections. In particular, there are conjunctive queries under set semantics for which no $\mathcal{L}(Q)$-exemplars exist for this reason [17].

Calculus of Equalities

We now reduce the containment problem of two queries in $\mathcal{L}(Q)$ to an implication problem of the two formulae derived from the `WHERE` clauses of the two queries. For this, we first define the equivalence of two formulae. Recall our general format of a conjunctive SQL query Q from before. For Q, let $V_Q := \{A_1, \ldots, A_n\}$ denote the attributes of R, which we view as *variables*, and let C_Q denote the *constant symbols* c that appear in the `WHERE` clause of Q. The *terms* of Q are $T_Q := V_Q \cup C_Q$. An *equality* τ for Q is either an attribute-attribute equality $A_r = A_s$ for $1 \leq r, s \leq n$ or an attribute-constant equality $A_r = c$ where $1 \leq r \leq n$ and $c \in C_Q$. A *formula* φ for Q is a conjunction of equalities for Q. We use $\mathcal{F}_0(Q)$ to denote the set of formulae for Q. A *conjunctive SQL query for* Q is a conjunctive SQL query of the form

$$\texttt{SELECT } A_{i_1}, \ldots, A_{i_m} \\ \texttt{FROM } R_1, \ldots, R_k \\ \texttt{WHERE } C'_1 \texttt{ AND} \ldots \texttt{AND } C'_o;$$

where $C'_1 \wedge \cdots \wedge C'_o \in \mathcal{F}_0(Q)$ is a formula for Q. We use $\mathcal{L}_0(Q)$ to denote the set of conjunctive SQL queries for Q. For $Q' \in \mathcal{L}_0(Q)$ let $\varphi_{Q'} = C'_1 \wedge \cdots \wedge C'_o$ denote the formula in $\mathcal{F}_0(Q)$ that forms the `WHERE` clause of Q'.

Example 2 *For the target query Q from Example 1,*

$$\texttt{c.cid=b.cid} \wedge \texttt{c.cid='c7'} \wedge \texttt{b.pid=p.pid} \wedge \texttt{b.date='3/9/14'}$$

denotes the formula φ_Q that forms its `WHERE` clause.

The *satisfaction* of a formula $\varphi \in \mathcal{F}_0(Q)$ by a tuple t over $R(A_1, \ldots, A_n)$, denoted by $\models_t \varphi$, is defined as follows:

- $\models_t A_r = A_s$ iff $t(A_r) = t(A_s)$,
- $\models_t A_r = c$ iff $t(A_r) = c$,
- $\models_t \varphi_1 \wedge \cdots \wedge \varphi_n$ iff for all $i = 1, \ldots, n$, $\models_t \varphi_i$.

For $\varphi, \psi \in \mathcal{F}_0(Q)$, φ *implies* ψ, denoted by $\varphi \rightarrow \psi$, iff every tuple t over $R(A_1, \ldots, A_n)$ that satisfies φ, satisfies ψ.

Example 3 *Let Q denote the target query from Example 1. The tuple*

(c0,Jimmy,p1,printer,c0,p1,3/9/2014,79)

over

R(c.cid,c.cname,p.pid,p.pname,b.cid,b.pid,b.date,b.cost)

does not satisfy φ_Q from Example 2, which is satisfied by the tuple

(c7,Jimmy,p1,printer,c7,p1,3/9/2014,79).

Let $\mathcal{F}(Q) \subseteq \mathcal{F}_0(Q)$ denote the formulae of $\mathcal{F}_0(Q)$ that are satisfiable. That is, those formulae $\varphi \in \mathcal{F}_0(Q)$ such that there is some tuple t over $R(A_1, \ldots, A_n)$ that satisfies φ. For example, `c.cid=b.cid` $\wedge$ `c.cid='c0'` $\wedge$ `b.cid='c7'` is unsatisfiable. Let $\mathcal{L}(Q) \subseteq \mathcal{L}_0(Q)$ denote the set of conjunctive SQL queries for Q where $\varphi_Q \in \mathcal{F}(Q)$ holds. We can now establish a correspondence between the containment of our queries and the implication of their associated formulae.

Lemma 3 *For $Q', Q'' \in \mathcal{L}(Q)$ we have $Q'' \subseteq_b Q'$ if and only if $\varphi_{Q''} \to \varphi_{Q'}$.*

Proof Based on our assumptions and the definition of $\mathcal{L}(Q)$, the `SELECT` and `FROM` clauses of Q'' and Q' are identical. Hence, containment $Q'' \subseteq_b Q'$ reduces to the problem whether every tuple that satisfies $\varphi_{Q''}$ also satisfies $\varphi_{Q'}$. That is, whether $\varphi_{Q''} \to \varphi_{Q'}$ holds.

Representative Tuples

For an $\mathcal{L}(Q)$-exemplar of Q we include some tuple t_0 that satisfies the equalities of Q, but does not satisfy any condition not entailed by Q. That way the tuple t_0 separates Q from any query Q' that is not a container of Q.

Definition 3 *For a conjunctive SQL query Q the tuple t over $R(A_1, \ldots, A_n)$* represents *$\varphi \in \mathcal{F}(Q)$ if and only if for all $\psi \in \mathcal{F}(Q)$, $\models_t \psi$ iff $\varphi \to \psi$ holds.* ∎

The existence of some tuple that represents some given $\varphi \in \mathcal{F}(Q)$ is guaranteed: attributes, that must carry the same value but not a constant from Q, are given the same unique value. For this reason, if $\varphi \to \psi$ does not hold, then the tuple t that represents φ can always be chosen such that $\models_t \psi$ does not hold.

Example 4 *For $\mathcal{D}$ with `C(cid,cname)`, `P(pid,pname)`, `B(cid,pid,date,cost)` and*

R(c.cid,c.cname,p.pid,p.pname,b.cid,b.pid,b.date,b.cost),

t_0=(c7,Jimmy,p1,printer,c7,p1,3/9/2014,79) represents φ_Q from Example 2, but it does not represent `c.cid=b.cid` $\wedge$ `c.cid='c7'` $\wedge$ `b.date='3/9/14'`. Indeed, t_0 also satisfies the equality `b.pid=p.pid` which is not implied by `c.cid=b.cid` $\wedge$ `c.cid='c7'` $\wedge$ `b.date='3/9/14'`.

Representative tuples resemble Armstrong databases [6, 8, 15]: for some tuple t that represents φ, deciding if φ implies ψ reduces to the problem of deciding whether t satisfies ψ.

We say the tuple t *represents* $Q' \in \mathcal{L}(Q)$ if and only if t represents $\varphi_{Q'} \in \mathcal{F}(Q)$. Any tuple t_0 representing Q will also be selected by any proper container of Q. Thus, additional tuples are required to separate Q from any of its proper containers. Evidently, it suffices to insert tuples that represent the minimal proper containers of Q.

Canonical Exemplars

We now construct exemplars from the tuples that represent the target query and its minimal proper containers.

Definition 4 *For a conjunctive query Q without self-joins let $\mathcal{M}(Q) = \{Q_1, \ldots, Q_p\}$. Let t_0 be a tuple that represents Q, and for $j = 1, \ldots, p$, let t_j denote a tuple that represents Q_j, such that the only values that occur in more than one tuple among $t_0, \ldots, t_j$ are constants from C_Q. For $i = 1, \ldots, k$, let $r_i = \pi_{R_i(A^i_1,\ldots,A^i_{n_i})}(\{t_0, t_1, \ldots, t_p\})$. We call $db_Q := \{r_1, \ldots, r_k\}$ a* canonical $\mathcal{L}(Q)$-exemplar *for Q.*

The following lemma establishes an important property of our canonical exemplar. We will use this property to show that a canonical exemplar is indeed an exemplar.

Lemma 4 *Let db_Q denote a canonical $\mathcal{L}(Q)$-exemplar for Q. For all $t \in \times_{i=1}^{k} r_i$, if $\models_t \varphi_{Q'}$ for some $Q' \in \mathcal{L}(Q)$, then $Q \subseteq_b Q'$ holds.*

Proof For all $A_i, A_j \in R$, all $c \in C_Q$, and all $t' \in \{t_0, t_1, \ldots, t_p\}$ the following hold: i) if $t'(A_i) = t'(A_j)$, then $t_0(A_i) = t_0(A_j)$, and ii) if $t'(A_i) = c$, then $t_0(A_i) = c$. Recall that the only values that occur in more than one tuple amongst $t_0, t_1, \ldots, t_p$ are the constants from C_Q. It follows that the cross product of $r_1, \ldots, r_k$ does not result in tuples that have equalities of attribute values that are not already present in t_0. That is, for each tuple $t \in \times_{i=1}^{k} r_i$ the following holds, too: if $t(A_i) = t(A_j)$, then $t_0(A_i) = t_0(A_j)$, and ii) if $t(A_i) = c \in C_Q$, then $t_0(A_i) = c$. Consequently, if $\models_t \varphi_{Q'}$, then $\models_{t_0} \varphi_{Q'}$, too. However, as t_0 represents φ_Q, it follows by Definition 3 that $\varphi_Q \to \varphi_{Q'}$ holds. This means that $Q \subseteq_b Q'$ by Lemma 3.

Soundness of Construction

Theorem 5 *Let Q be a conjunctive SQL query without self-joins. For all $Q' \in \mathcal{L}(Q)$, $Q \equiv_b Q'$ if and only if $Q(db_Q) = Q'(db_Q)$.*

Proof It follows from the definition of $\equiv_b$ that $Q \equiv_b Q'$ entails $Q(db_Q) = Q'(db_Q)$.

Assume now that $Q \not\equiv_b Q'$. That is, either 1) $Q \not\subseteq_b Q'$, or 2) $Q \subsetneq_b Q'$. Case 1) means that $\varphi_Q \to \varphi_{Q'}$ does not hold, by Lemma 3. As $t_0 \in r_1 \times \ldots \times r_k$ represents Q it follows by Definition 3 that $\models_{t_0} \varphi_Q$. Lemma 4 shows that for all $t \in r_1 \times \ldots \times r_k$, $\not\models_t \varphi_{Q'}$. Therefore, $Q(db_Q) \neq \emptyset = Q'(db_Q)$. Case 2) means that there is some $Q_i \in \mathcal{M}(Q)$ such that $Q_i \subseteq_b Q'$, by definition of the minimal proper containers Q_i of Q. Consequently, $\varphi_{Q_i} \to \varphi_{Q'}$ holds, by Lemma 3. As $t_i \in r_1 \times \ldots \times r_k$ represents Q_i it follows by Definition 3 that $\models_{t_i} \varphi_{Q'}$. However, $\models_{t_i} \varphi_Q$ does not hold. Since $Q \subseteq_b Q'$, $\varphi_Q \to \varphi_{Q'}$ holds. Therefore,

$$\{t \in r_1 \times \cdots \times r_k \mid \models_t \varphi_Q\} \subsetneq_b \{t \in r_1 \times \cdots \times r_k \mid \models_t \varphi_{Q'}\}.$$

Consequently, $Q(db_Q) \neq Q'(db_Q)$ by Lemma 2.

Corollary 6 *For a conjunctive SQL query Q without self-joins, every canonical $\mathcal{L}(Q)$-exemplar for Q is an $\mathcal{L}(Q)$-exemplar for Q.* ▮

5.3 Computation of Canonical Exemplars

The construction of a canonical exemplar is founded on tuples that represent the target query Q and its minimal proper containers. Each query $Q' \in \mathcal{L}(Q)$ partitions the set of terms T_Q of Q into equivalence classes, induced by the equalities in the `WHERE` clause of Q'. Formally, for $\tau, \tau' \in T_Q$, we define $\tau \equiv_{Q'} \tau'$ if and only if $\varphi_{Q'} \rightarrow \tau = \tau'$. Evidently, $\equiv_{Q'}$ defines an equivalence relation over the terms T_Q of Q. Let $[\tau]_{\equiv_{Q'}}$ denote the equivalence class of $\tau \in T_Q$, i.e., $\{\tau' \in T_Q \mid \tau \equiv_{Q'} \tau'\}$. Finally, let $S_{\equiv_{Q'}}$ denote the quotient of T_Q with respect to $\equiv_{Q'}$, that is, $S_{\equiv_{Q'}} = \{[\tau]_{\equiv_{Q'}} \mid \tau \in T_Q\}$.

Algorithm 1 shows the overall construction of a canonical $\mathcal{L}(Q)$-exemplar for Q. Line 1 computes the quotient $S_{\equiv_{Q_0}}$ of the input query Q. Starting with the partition of singleton subsets of T_Q the elements of the partition are merged as the equalities in the `WHERE` clause are scanned one by one.

Lines 2-9 compute the set $\mathcal{M}(Q)$ of minimal proper containers for Q, represented by their quotients $S_{\equiv_{Q_i}}$. Each $S_{\equiv_{Q_i}}$ is derived from $S_{\equiv_Q}$ by replacing one element $S \in S_{\equiv_Q}$ by an element s from one of its bi-partitions S', i.e., a partition into two non-empty subsets. All-Bi-Partitions(S) returns the set of all bi-partitions of a set S [20].

Lines 10-12 compute tuples that represent Q and its minimal proper containers $Q_1, \ldots, Q_p$. The only values that occur in more than one representative tuple are the constants from Q. A representative tuple can easily be constructed from a quotient $S_{\equiv_{Q'}}$: two attributes A_r and A_s receive the same value iff they belong to the same equivalence class. This value must equal the constant if the latter is also an element of the same equivalence class. An equivalence class contains two constants iff $\varphi_{Q'}$ is unsatisfiable. In practice, the algorithm will terminate in this case and return nothing.

Lines 13-15 compute a canonical $\mathcal{L}(Q)$-exemplar db_Q as projections $r_1, \ldots, r_k$ of $db = \{t_0, \ldots, t_p\}$ to the participating relation schemata $R_i(A^i_1, \ldots, A^i_{n_i})$ for $i = 1, \ldots, k$.

5.4 Complexity

The complexity of Algorithm 1 is dominated by the computation of all bi-partitions for each element of Q's quotient. The number of partitions of an n-element set into k non-empty subsets is the *Stirling number of the second kind*, denoted by $S(n, k)$. Here, the relevant case is $k = 2$ where $S(n, 2) = 2^{n-1} - 1$ and the maximum number of minimal proper containers is attained when $S_{\equiv_Q} = \{T_Q\}$. The computation of the bi-partitions only starts when T_Q contains at most one constant. Using our notation $R(A_1, \ldots, A_n)$, T_Q contains at most $n + 1$ elements and there can be at most $2^n - 1$ minimal proper containers and 2^n representative tuples. As query equivalence under set semantics is known to be NP-complete, it is elusive to find a *PTIME*-algorithm.

Proposition 7 *For each conjunctive SQL query Q that has a total of n attributes in the relation schemata that occur in its* `FROM` *clause, Algorithm 1 returns a canonical $\mathcal{L}(Q)$-exemplar for Q in time and space that is in $\mathcal{O}(2^n)$.* ▮

Let Q be a conjunctive SQL query with a total number n of participating attributes and a total number k of participating constants. Then the number of queries that an $\mathcal{L}(Q)$-exemplar can separate from Q is the $m = n + k$-th Bell number B_m, which counts

Algorithm 1 Computation of Canonical Exemplar

```
Require: Conjunctive SQL Query $Q$ without Self-Joins
Ensure: $\mathcal{L}(Q)$-canonical exemplar for $Q$
 1: $S_{\equiv_{Q_0}} \leftarrow$ QUOTIENT$(Q)$
 2: $\mathcal{M}(Q) \leftarrow \emptyset$
 3: for all $S \in S_{\equiv_Q}$ do
 4:     $S' \leftarrow$ ALL-BI-PARTITIONS$(S)$
 5:     for all $s \in S'$ do
 6:         $\mathcal{M}(Q) \leftarrow \mathcal{M}(Q) \cup \{(S_{\equiv_Q} - \{S\}) \cup \{s\}\}$
 7:     end for
 8: end for
 9: $\mathcal{M}(Q) = \{S_{\equiv_{Q_1}}, \dots, S_{\equiv_{Q_p}}\}$
10: for $i = 0, \dots, p$ do $t_i \leftarrow$ REPRESENTATIVE$(S_{\equiv_{Q_i}})$
11: end for;
12: $db \leftarrow \{t_0, t_1, \dots, t_p\}$
13: for $i = 1, \dots, k$ do $r_i \leftarrow \pi_{R_i.A^i_1,\dots,R_i.A^i_{n_i}}(db)$
14: end for;
15: $db_Q \leftarrow \{r_1, \dots, r_k\}$
16: Return$(db_Q)$
```

the number of partitions of an m-element set. These may also include queries with inconsistent conditions, which we assume to be not equivalent to the target query. The m-th Bell number can be computed with the following recurrence relation $B_m = \sum_{j=0}^{m-1} B_j \cdot \binom{m-1}{j}$ where $B_0 := 1 = B_1$. As can be observed from Table 1 the number of pairwise non-equivalent queries that an exemplar can separate from Q is at least exponential in the size of the exemplar. For example, for $n = 14$ and $k = 1$ the maximum-sized canonical $\mathcal{L}(Q)$-exemplar has $16,384$ tuples and can separate $B_{15} = 1,382,958,545$ queries from Q, all pairwise non-equivalent. That is, the $\mathcal{L}(Q)$-exemplar can distinguish more than $84,000$ times as many semantically different queries as its own size.

Proposition 8 *Algorithm 1 returns $\mathcal{L}(Q)$-exemplars db_Q that can separate a number of pairwise non-equivalent queries from Q that grows at least exponentially in the size of db_Q.* ∎

6 Show Case

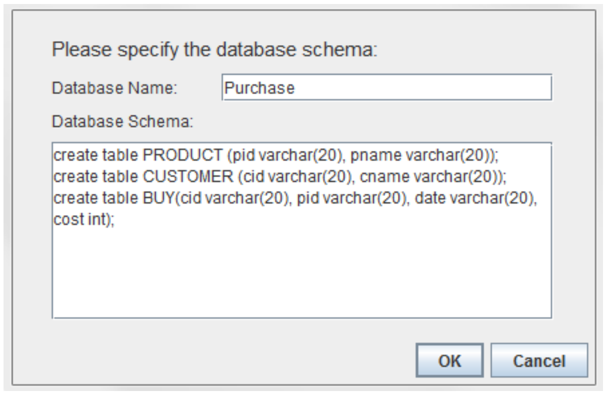
This section illustrates the use of LEQCTER in its primary application area as an automated lector. The case shows how LECQTER exploits its ability to test query equivalence in assisting trainees in the process of learning how to write sound conjunctive SQL queries, and in assisting trainers to provide automated feedback and marks from which trainees can confidently learn. As show case we consider the schema $\mathcal{D}$ and target query Q from Example 1.

n	s_n	q_n	q_n/s_n
5	16	52	$> 3 > 2^1$
6	32	203	$> 6 > 2^2$
7	64	877	$> 13 > 2^4$
8	128	4140	$> 32 = 2^5$
9	256	21,147	$> 82 > 2^7$
10	512	115,975	$> 226 > 2^8$
11	1024	678,570	$> 662 > 2^{10}$
12	2048	4,213,597	$> 2057 > 2^{11}$
13	4096	27,644,437	$> 6749 > 2^{13}$
14	8192	190,899,322	$> 23,303 > 2^{15}$
15	16384	1,382,958,545	$> 84,409 > 2^{16}$

Table 1: Maximum size s_n of exemplar, number q_n of non-equivalent queries the exemplar can separate, and the growth s_n/q_n for the number $n = 5, \ldots, 15$ of terms in the target query

6.1 Specify the Database Schema

Initially, a trainer uses LECQTER to create an underlying database schema in MySQL. As can be seen in Figure 1, name and database schema are entered in the *startGUI* window. When the "OK" button is clicked, the database schema named "***Purchase***" is created on the MySQL database server by LECQTER.


Please specify the database schema:

Database Name: Purchase

Database Schema:

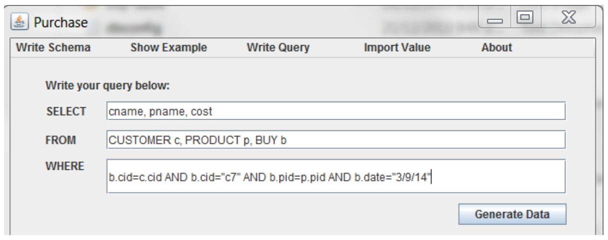
```
create table PRODUCT (pid varchar(20), pname varchar(20));
create table CUSTOMER (cid varchar(20), cname varchar(20));
create table BUY(cid varchar(20), pid varchar(20), date varchar(20),
cost int);
```

Figure 1: Creation of the Database Schema

6.2 Target Query

The tutor enters the target query in the *queryGUI* window, as shown in Figure 2, and clicks on the "Generate Data" button.

This invokes the construction of a canonical exemplar by *Lecqter*, which populates the database ***Purchase***. For instance, Figure 3 shows the ***Purchase***-table BUY of the exemplar created.



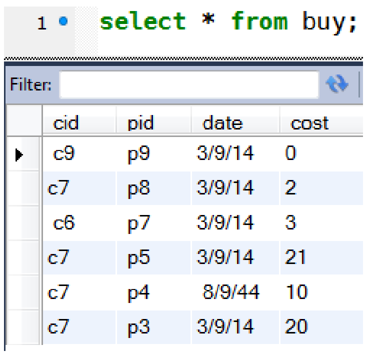
Write your query below:

SELECT cname, pname, cost

FROM CUSTOMER c, PRODUCT p, BUY b

WHERE b.cid=c.cid AND b.cid="c7" AND b.pid=p.pid AND b.date="3/9/14"

Figure 2: Submitting the Target Query



```
select * from buy;
```

cid	pid	date	cost
c9	p9	3/9/14	0
c7	p8	3/9/14	2
c6	p7	3/9/14	3
c7	p5	3/9/14	21
c7	p4	8/9/44	10
c7	p3	3/9/14	20

Figure 3: Table BUY of the ***Purchase*** database

Once an exemplar has been constructed the text box for entering equalities is cleared while the `SELECT` and `FROM` clauses remain the same as in the target query and are not allowed to be changed for the current session. Additionally, the text of "Generate Data" button becomes "Run Query".

User Queries

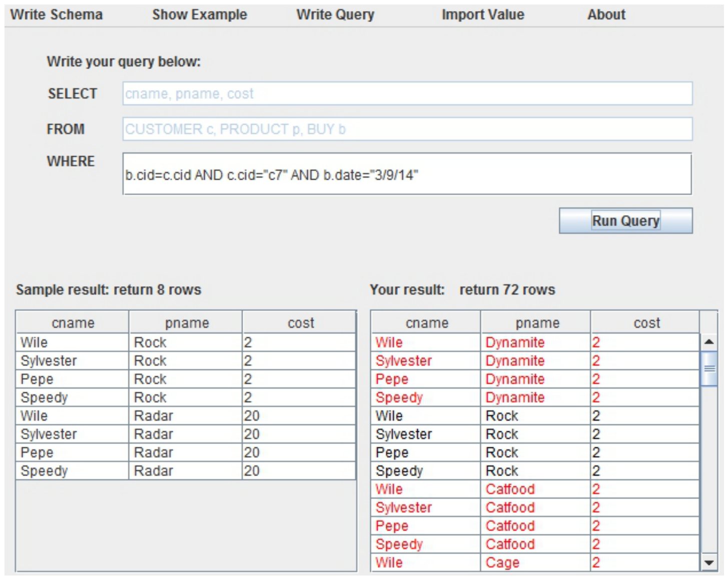
Once LECQTER has created an $\mathcal{L}(Q)$-exemplar in MySQL based on the target query Q and database schema $\mathcal{D}$, trainees submit `WHERE` clauses that shall match the description of Q. Trainee A does not specify the equality `b.pid = p.pid`. Results of A's query are shown in the bottom right with tuples not in Q being highlighted in red, as shown in Figure 4.

Trainee B submits a query with

`WHERE b.pid=c.cid AND c.cid='c7' AND b.pid=c.cid AND b.date='3/9/14'`.

This query illustrates that typos can easily lead to incorrect results. The empty result to B's query is shown in Figure 5, it originates from the incorrect equality `b.pid=c.cid`.

It can be seen from the screen shots that tuples in the results of user queries that are not in the result of the target query are highlighted in red color. Furthermore, the number of rows returned is presented as well. This makes it easy for the tutor and trainee to spot any differences in the trainee's results from the target results, even when the results are large. The differences in the results may provide useful information for the trainees to figure out what is wrong with their queries, so it is helpful in trial-and-error learning, too.



Write your query below:

SELECT cname, pname, cost

FROM CUSTOMER c, PRODUCT p, BUY b

WHERE b.cid=c.cid AND c.cid="c7" AND b.date="3/9/14"

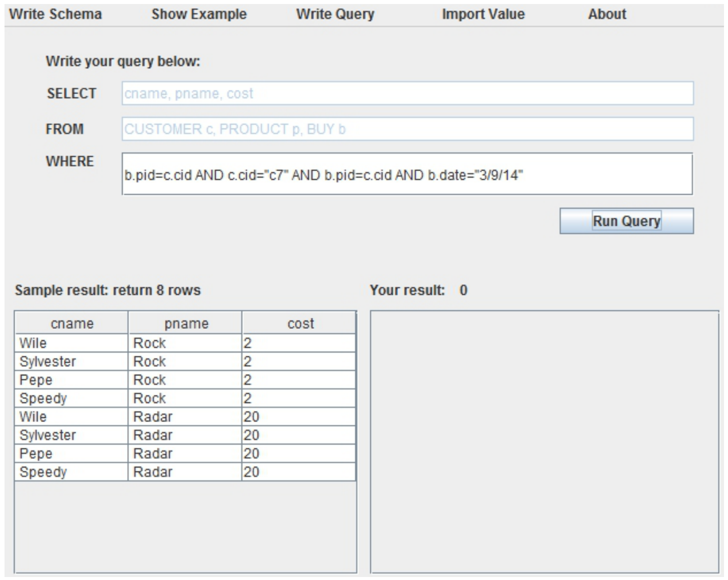
Sample result: return 8 rows

cname	pname	cost
Wile	Rock	2
Sylvester	Rock	2
Pepe	Rock	2
Speedy	Rock	2
Wile	Radar	20
Sylvester	Radar	20
Pepe	Radar	20
Speedy	Radar	20

Your result: return 72 rows

cname	pname	cost
Wile	Dynamite	2
Sylvester	Dynamite	2
Pepe	Dynamite	2
Speedy	Dynamite	2
Wile	Rock	2
Sylvester	Rock	2
Pepe	Rock	2
Speedy	Rock	2
Wile	Catfood	2
Sylvester	Catfood	2
Pepe	Catfood	2
Speedy	Catfood	2
Wile	Cage	2

Figure 4: User Query Returning More Tuples



Write your query below:

SELECT cname, pname, cost

FROM CUSTOMER c, PRODUCT p, BUY b

WHERE b.pid=c.cid AND c.cid="c7" AND b.pid=c.cid AND b.date="3/9/14"

Sample result: return 8 rows

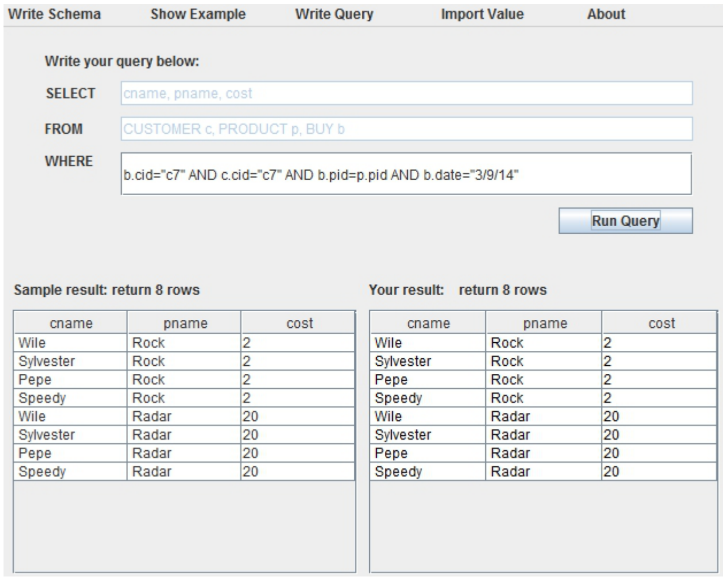
cname	pname	cost
Wile	Rock	2
Sylvester	Rock	2
Pepe	Rock	2
Speedy	Rock	2
Wile	Radar	20
Sylvester	Radar	20
Pepe	Radar	20
Speedy	Radar	20

Your result: 0

Figure 5: User Query Returning Empty Answer

In Figure 6 trainee C submits the `WHERE` clause `b.pid=‘c7’ AND c.cid=‘c7’ AND b.pid=c.cid AND b.date=‘3/9/14’`.

As the answers match the target, trainer and trainee do not need to inspect the trainee query to conclude its correctness. This is not the case for queries evaluated on any database, but the distinguishing feature of Lecqter.



Write your query below:

SELECT cname, pname, cost

FROM CUSTOMER c, PRODUCT p, BUY b

WHERE b.cid="c7" AND c.cid="c7" AND b.pid=p.pid AND b.date="3/9/14"

Sample result: return 8 rows

cname	pname	cost
Wile	Rock	2
Sylvester	Rock	2
Pepe	Rock	2
Speedy	Rock	2
Wile	Radar	20
Sylvester	Radar	20
Pepe	Radar	20
Speedy	Radar	20

Your result: return 8 rows

cname	pname	cost
Wile	Rock	2
Sylvester	Rock	2
Pepe	Rock	2
Speedy	Rock	2
Wile	Radar	20
Sylvester	Radar	20
Pepe	Radar	20
Speedy	Radar	20

Figure 6: Sound User Query

Lecqter is available at `slink.foiks.org/Lecqter.zip` for download. It can be run on Windows 7, and access to a MySQL server can easily be configured.

6.3 Additional Features

It is noted again that Lecqter abandons the construction in case that the formula φ_Q is unsatisfiable. For instance, a target query Q with inconsistent equalities such as `p.pid = b.pid and p.pid = 0 and b.pid = 1` results in an equivalence class {p.pid,b.pid,0,1} with two constants, at which point Lecqter stops.

Users can specify values to populate representative tuples during construction time. Users can click the “Import Value” button and select a text file with a comma-separated list of values. The name of the text file matches that of the attribute. Once the pre-specified list of values has been exhausted, new synthetic values are generated as required.

Users can change the configuration to access MySQL server by editing the text file “dbconfig.txt”, which is located in the same folder Lecqter. The user name, password, and IP address are set to: “admin|admin|127.0.0.1:3306”.

7 Experimental Results

The worst-case time and space complexity of constructing canonical $\mathcal{L}(Q)$-exemplars is exponential in the total number of participating attributes. We will now establish empirical evidence that $\mathcal{L}(Q)$-exemplars can be generated efficiently for the purpose of learning how to write sound conjunctive SQL queries. Furthermore, we establish evidence that the evaluation of user queries on $\mathcal{L}(Q)$-exemplars is efficient, too. Together, the experiments show that $\mathcal{L}(Q)$-exemplars are a practical tool for learning how to write sound conjunctive SQL queries, as they can be generated efficiently and are able to efficiently separate at least an exponential number of non-equivalent queries from the target query.

Experiments were carried out on a MySQL 5.6 database server running on an Asus laptop, with 2.5 GHz i5-2450M Dual-Core processor and 4GB RAM, on a Windows 7 Home Premium operating system. We analyzed four different classes of queries regarding their performance in constructing canonical $\mathcal{L}(Q)$-exemplars with Algorithm 1.

7.1 Construction of Exemplars

First, we present four different classes of queries for which analyze the performance of constructing canonical $\mathcal{L}(Q)$-exemplars with Algorithm 1, as well as its size. Two of these test cases illustrate extreme behavior: one of these cases produces exemplars of constant size in almost constant time with increasing input size, while the other case produces exemplars of size exponential in time exponential in the input size. The third test case produces exemplars of size linear in time linear in the input size. Each of these three test cases follows a particular pattern that determines the performance of constructing exemplars. Apart from revealing the actual time it takes to construct an exemplar, our experiments therefore also show what behavior to expect when these patterns occur in a real-life target query. The fourth and final test case creates equalities randomly.

Constant

Target queries Q_n^c in this test case have the format

$$\begin{array}{ll} \texttt{SELECT} & * \\ \texttt{FROM} & R \\ \texttt{WHERE} & a_1 = 0 \end{array}$$

where $R(A_1, \ldots, A_n)$ is the only present relation schema. For this experiment, n varied from 1 up to 50. For each n, there is one tuple that represents Q_n^c and one tuple that represents the only minimal proper container of Q_n^c. Therefore, the size of the resulting test database is constant for all n, and the times to construct a canonical exemplar are illustrated in Figure 7. Minimum, average and maximum times were taken over 100 runs for each n.

Linear

Target queries Q_n^l in this test case have the format

$$\begin{array}{ll} \texttt{SELECT} & * \\ \texttt{FROM} & R_1, \ldots, R_n \\ \texttt{WHERE} & R_1.A_1 = R_2.A_1 \texttt{ AND } R_2.A_2 = R_3.A_2 \texttt{ AND } \cdots \texttt{ AND} \\ & R_{n-1}.A_s = R_n.A_s \end{array}$$

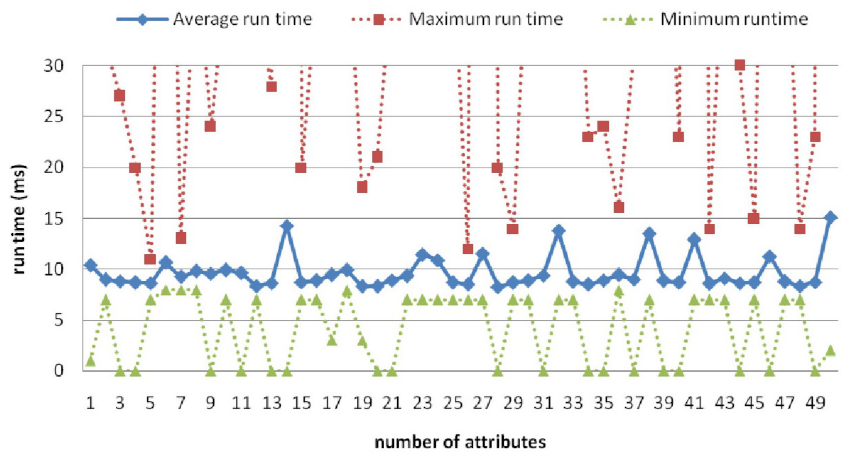


Figure 7: Construction Times for **Constant**

where $s = 1$ when n is even, and $s = 2$ when n is odd, and $R_1(A_1, A_2), \ldots, R_n(A_1, A_2)$ form the underlying database schema. For this experiment, n also varied from 1 up to 50. For each n, there is one tuple that represents Q^l_n and one tuple that represents each of the $n-1$ minimal proper containers of Q^l_n. Therefore, the size of the resulting exemplar is n, and the times to construct a canonical exemplar are illustrated in Figure 8. Minimum, average and maximum times were taken over 100 runs for each n.

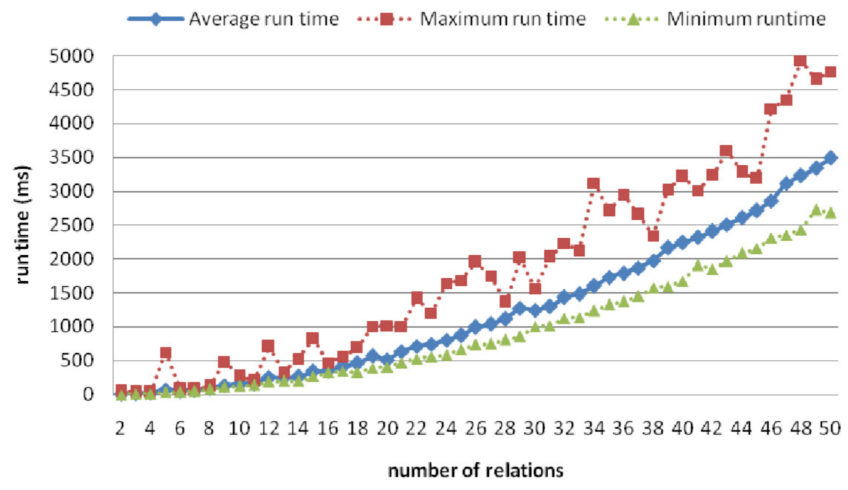


Figure 8: Construction Times for **Linear**

Exponential

Target queries Q^e_n in this test case have the format

```
SELECT  *
FROM    $R$
WHERE   $A_1 = A_2$ AND $A_2 = A_3$
        AND $\cdots$ AND $A_{n-1} = A_n$
```

where $R(A_1, \dots, A_n)$ is the only present relation schema. For this experiment, n varied from 2 up to 15 due to memory limits. For each n, there are one tuple that represents Q_n^e and for each of the $2^{n-1} - 1$ minimal proper containers of Q_n^e one representative tuple. Therefore, the size of the resulting exemplar is 2^{n-1}, and the times to construct a canonical exemplar are illustrated in Figure 9. Minimum, average and maximum times were taken over 100 runs for each n.

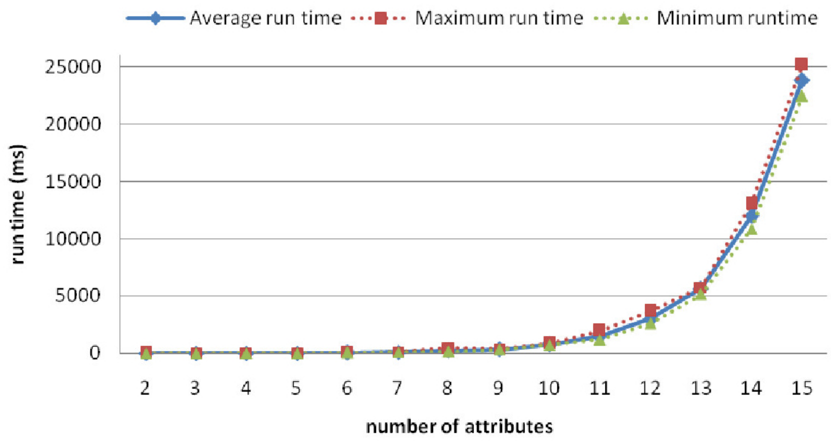


Figure 9: Construction Times for **Exponential**

Random

Target queries Q_n^r in this test case have the format

```
SELECT  *
FROM    R
WHERE   ...;
```

with $R(A_1, \dots, A_n)$ being the only relation schema, and the equalities in the `WHERE` clause are created randomly. For this experiment, n varied from 3 up to 15 due to memory limits. Minimum, average and maximum times were taken over 1,000 runs for each n. For each n, the algorithm randomly selects in each run the number of equalities that will form the `WHERE` clause. Four different experiments were conducted, where the maximum number of equalities to be randomly selected in each run for each n had an upper bound of $k \cdot n$ where $k \in \{0.25, 0.5, 0.75, 1\}$, respectively. The sizes of the exemplars are shown in Figures 10, 11, 12, and 13, respectively.

The figures confirm convincingly our intuition that the number of equalities selected is proportional to the size of the canonical exemplars constructed, and therefore also the time required to construct them.

Figure 13 for $k = 1$, in particular, illustrates a significant difference between the average and maximum sizes of exemplars when equalities are selected randomly. While the maximum size of exemplars for Q_n^r coincides with the size of the exemplars for Q_n^e, the average size is significantly smaller. Note that the minimum sizes for Q_n^r are zero,

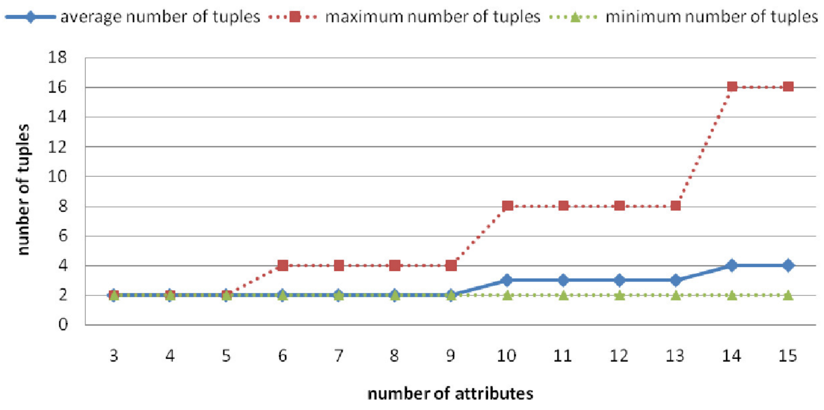


Figure 10: Exemplar Sizes for **Random**, $k = 0.25$

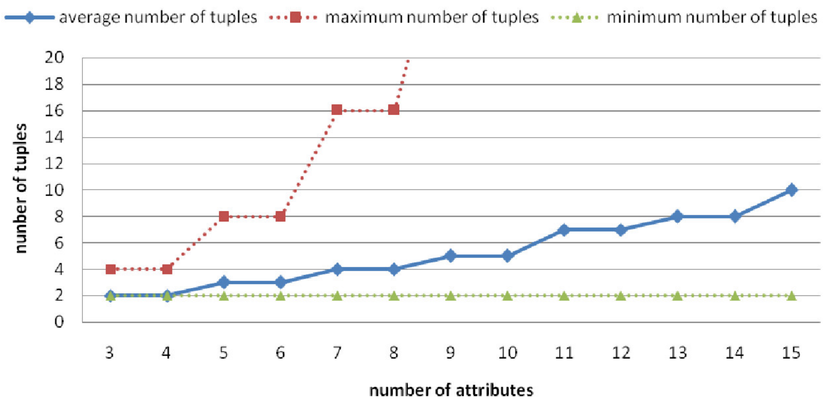


Figure 11: Exemplar Sizes for **Random**, $k = 0.5$

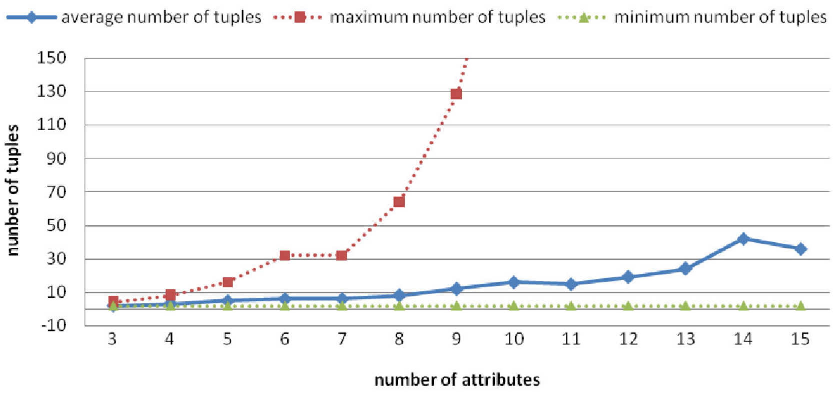


Figure 12: Exemplar Sizes for **Random**, $k = 0.75$

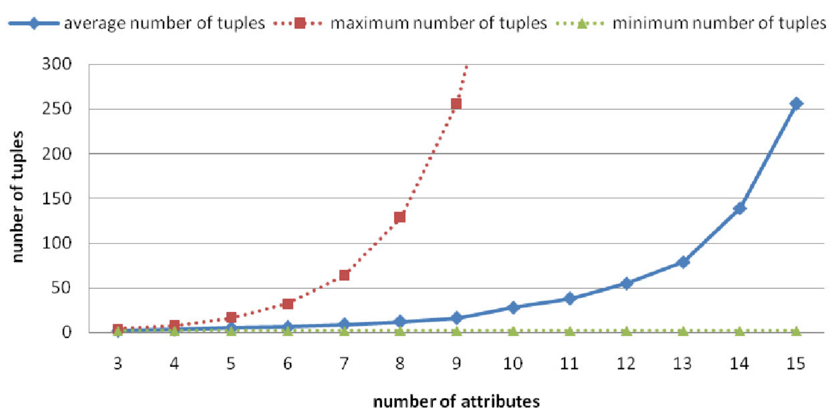


Figure 13: Exemplar Sizes for **Random**, $k = 1$

indicating that the number of runs and number of equalities are sufficient to generate at least one inconsistent input.

Figure 14 shows the minimum, average and maximum times taken for each n to create a canonical $\mathcal{L}(Q)$-exemplar given Q^e_n. The figure illustrates the significant difference between the average and maximum times required to construct canonical exemplars when equalities are selected randomly.

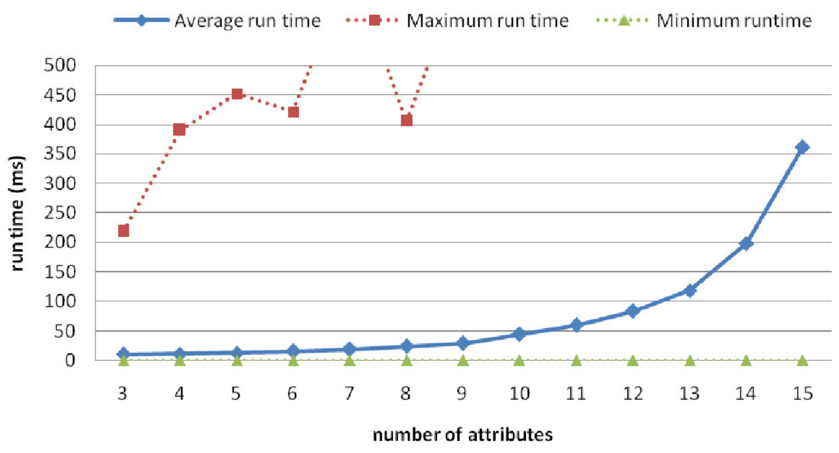


Figure 14: Construction Times for **Random**, $k = 1$

For instance, the average and maximum sizes of the canonical exemplars for Q^r_{15} are listed in Table 2, together with the average and maximum times to construct them. The tables illustrate the impact of the maximum number of equalities permitted, and the significant differences in size and time between the average and maximum cases.

So far, all experiments in the class **Random** were conducted with a fair chance of selecting an attribute-attribute equality and attribute-constant equality, respectively. Intuitively, the higher the probability of selecting attribute-attribute equalities, the larger the size of an exemplar will be. Attribute-constant equalities may force different constants to be the same, in which case the input is inconsistent and the size and time measured are both zero. Several attribute-attribute equalities increase the chance of a

k	0.25	0.5	0.75	1
avg size	4	10	30	256
avg time	10	21	56	361
max size	16	256	2,048	16,384
max time	78	546	3,838	22,169

Table 2: Average and maximum sizes of canonical exemplars and times in ms to construct them for Q^r_{15} and $k = 0.25, 0.5, 0.75, 1$

p	0.25	0.5	0.75
avg	118	256	330
max	8,193	16,384	16,384

Table 3: Average and maximum sizes of canonical exemplars for Q^r_{15} with $k = 1$ and different probabilities p on attribute-attribute equalities

large equivalence class, resulting in larger exemplars. We have confirmed this intuition by running two further experiments in the class **Random** where the probability p of selecting an attribute-attribute equality was 0.25 and 0.75, respectively. Each experiment was repeated 1,000 times for each input size n. For each n, a constant could be chosen between 1 and n. While this margin may be relatively small, the outcomes already confirm the intuition above, thereby suggesting a stronger trend with bigger margins. Figures 15, 13 and 16 show the sizes of exemplars constructed for $p = 0.25$, $p = 0.5$ and $p = 0.75$, respectively.

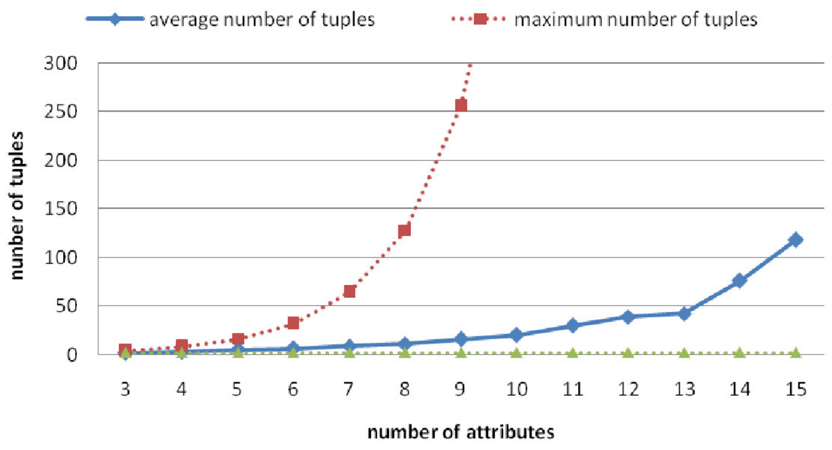


Figure 15: Exemplar Sizes for **Random**, $k = 1$, $p = 0.25$

Average and maximum sizes of the canonical exemplars for Q^r_{15} with $k = 1$ and $p \in \{0.25, 0.5, 0.75\}$ are listed in Table 3. The table confirms that the higher the probability of selecting attribute-attribute equalities the larger the exemplars become. Table 3 also shows that the maximum size was not attained in any of the 1,000 runs with $p = 0.25$.

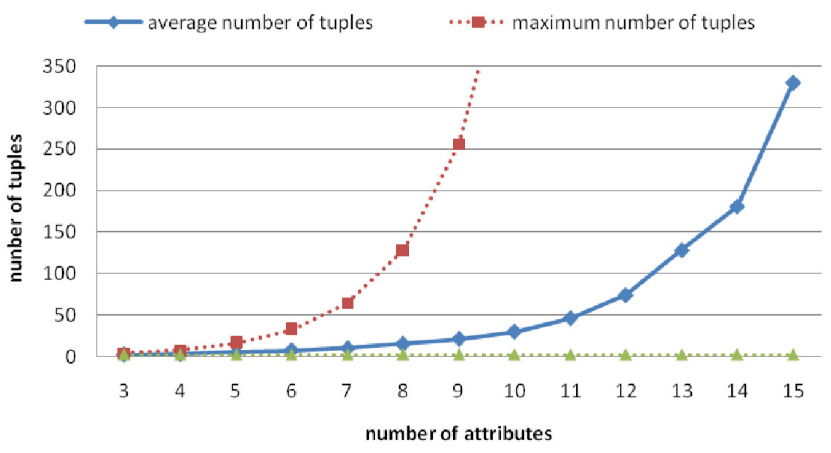


Figure 16: Exemplar Sizes for **Random**, $k = 1$, $p = 0.75$

7.2 Query Evaluation on Exemplars

Another important contributing factor to the performance of LECQTER concerns the evaluation of queries on the exemplars constructed. It is important that users perceive the evaluation of their queries as efficient. Otherwise, the application might not be perceived as helpful.

For this purpose, some experiments were conducted regarding the evaluation of queries on the exemplars that were constructed for each of the first three test cases. For each case, we evaluated all queries on the largest exemplar generated.

For each case, the number of tuples returned by each query as well as the time taken to return these tuples were recorded. In what follows, the results of these experiments are presented and discussed for each test case in turn.

Constant

Here, we evaluated for each $n = 1, \ldots, 50$, the target query Q^c_n on its exemplar $db_{Q^c_n}$. In each case, a single tuple was therefore returned as the answer. Figure 17 illustrates the minimum, average and maximum runtime to evaluate each of the 50 queries. These were taken out of 500 runs for each query. The average time for evaluating each query is almost identical, that is, between 4 and 6ms.

Linear

Here, we evaluated for each $n = 1, \ldots, 50$, the target query Q^l_n on the fixed exemplar $db_{Q^l_{50}}$. Therefore, $51 - n$ tuples were returned as the answer for the query Q^l_n, for $n = 1, \ldots, 50$. Figure 18 illustrates the minimum, average and maximum runtime to evaluate each of the 50 queries. These were taken out of 500 runs for each query. For example, the evaluation of Q^l_{50} takes less than 80ms.

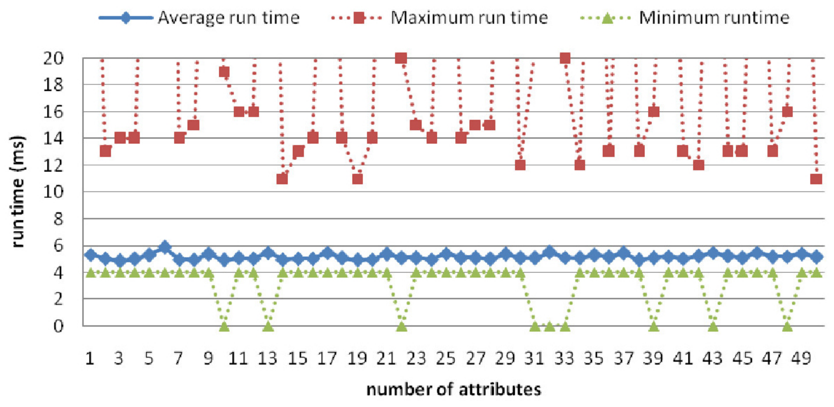


Figure 17: Query times for **Constant** [500]

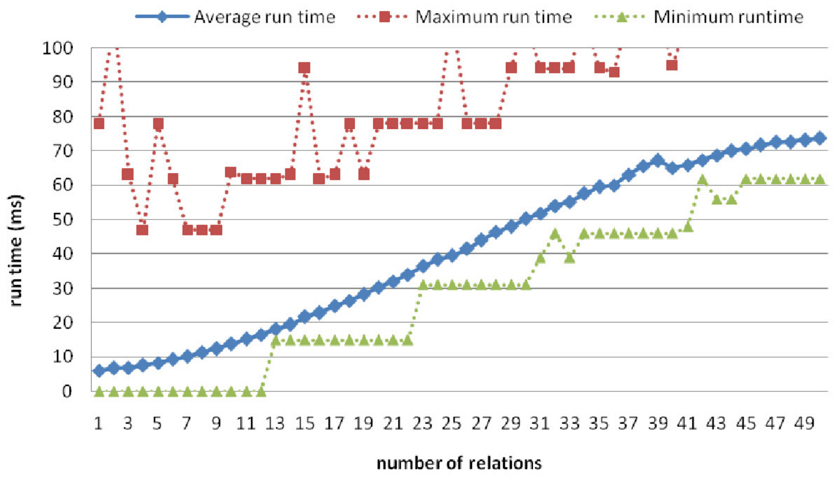


Figure 18: Query times for **Linear**

Exponential

Here, we evaluated for each $n = 1, \ldots, 15$, the following query Q_n on the fixed exemplar $db_{Q^e_{15}}$:

$$\begin{array}{ll} \texttt{SELECT} & * \\ \texttt{FROM} & R \\ \texttt{WHERE} & A_1 = A_2 \texttt{ AND } A_2 = A_3 \\ & \texttt{AND } \cdots \texttt{ AND } A_{i-1} = A_i \end{array}$$

Therefore, 2^{15-n} tuples were returned as the answer for the query Q_i, for $n = 1, \ldots, 15$. Figure 19 illustrates the minimum, average and maximum runtime to evaluate each of the 15 queries. These were taken out of 500 runs for each query. For example, the average evaluation of Q_1 takes approximately 70ms, as it returns all 16,384 tuples. Subsequently, the average evaluation time stays at approximately 20ms for the remaining queries.

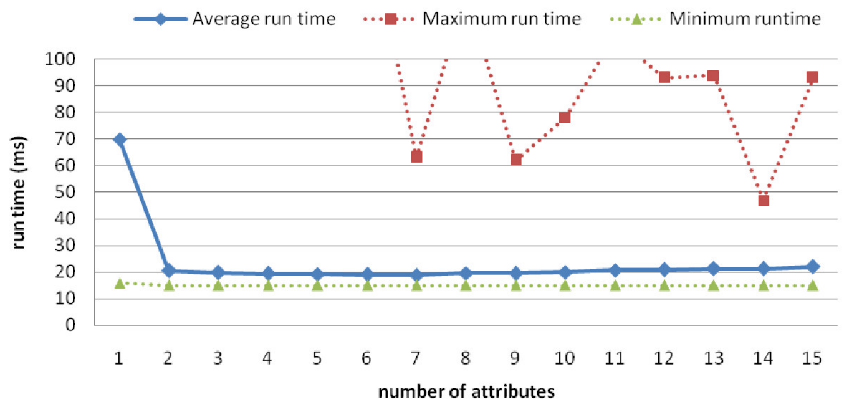


Figure 19: Query times for **Exponential**

7.3 Summary

Considering the first series of experiments, we conclude that the construction of canonical exemplars with LECQTER is efficient, in particular when considering the actual times to create them and their purpose for learning how to write sound conjunctive SQL queries. Here, the number of participating attributes and constants should not be chosen too large to remain illustrative for teaching purposes. While the extreme cases require dramatically more time and space, even these cases are feasible. It is also unlikely that these extreme cases occur in practice where the number of equalities can be expected to be rather modest relative to the size of the underlying schema.

Considering the second series of experiments, we conclude that the evaluation of queries on the exemplars constructed by LECQTER is efficient. Users of the application will therefore obtain feedback to their queries efficiently. The feedback will tell them with full certainty whether their queries are semantically correct or incorrect. In case the user query is incorrect, the difference in its result to the target answer is illustrated effectively.

8 Demo Databases

The example from Section 2.3 has shown that exemplars do not always exist for conjunctive queries when set semantics is used, as already observed in [17]. Our results have shown that for real conjunctive queries, namely conjunctive SQL queries or conjunctive queries under bag semantics, exemplars can be constructed efficiently in practice. Nevertheless, one may still wonder how the situation can be handled when trainees are exposed to `SELECT DISTINCT` queries. In this section, one possible approach is outlined.

We denote by Q_d the query obtained from the conjunctive SQL query Q by adding `DISTINCT` to its `SELECT` clause. What Section 2.3 has shown is that there are conjunctive target queries Q_d for which no database can exist on which all queries in $\mathcal{L}(Q_d)$, that are not equivalent to Q_d, produce answers that are different from those of Q_d. We suggest in this case to construct for each $Q'_d \in \mathcal{L}(Q_d)$ a *demo database* $demo_{Q'_d}$ from our exemplar db_Q that demonstrates either the equivalence of Q_d and Q'_d by producing the same answers on $demo_{Q'_d}$, or their difference by producing different answers on $demo_{Q'_d}$. Therefore, db_Q serves as a universal basis for constructing for each $Q'_d \in \mathcal{L}(Q_d)$, a demo database that illustrates the (non-)equivalence between Q_d and Q'_d.

For Q_d we construct an $\mathcal{L}(Q)$-exemplar $db_Q = \{r_1, \ldots, r_k\}$ for the conjunctive SQL query Q as before. For an arbitrary query $Q'_d \in \mathcal{L}(Q_d)$ we first evaluate $Q'(db_Q)$. If $Q'(db_Q) = Q(db_Q)$, then $demo_{Q'_d}$ consists of the relations obtained from projecting t_0 down to each of the participating relation schemata. In that case, both $Q'_d(demo_{Q'_d})$ and $Q_d(demo_{Q'_d})$ return $\pi_{A_{i_1},\ldots,A_{i_m}}(t_0)$. Otherwise, we distinguish the following two cases. The first case is where t_0 is not in $\sigma_{\varphi_{Q'}}(r_1 \times \cdots \times r_k)$. Then $demo_{Q'_d}$ consists of the relations obtained from projecting t_0 down to each of the participating relation schemata. In that case, $Q'_d(demo_{Q'_d})$ returns the empty answer and $Q_d(demo_{Q'_d})$ returns $\pi_{A_{i_1},\ldots,A_{i_m}}(t_0)$. The second and remaining case is where there is some $t_i \in \sigma_{\varphi_{Q'}}(r_1 \times \cdots \times r_k) - \sigma_{\varphi_Q}(r_1 \times \cdots \times r_k)$ that represents the minimal proper container $Q_i \in \mathcal{M}(Q)$. In that case, $demo_{Q'_d}$ consists of the relations obtained from projecting t_i down to each of the participating relation schemata. Consequently, $Q'_d(demo_{Q'_d})$ returns $\pi_{A_{i_1},\ldots,A_{i_m}}(t_i)$ and $Q_d(demo_{Q'_d})$ returns the empty set. We thus obtain the following result that offers a solution to the class of conjunctive queries without self-joins under set semantics.

Theorem 9 *For every conjunctive query Q without self-joins and every conjunctive query $Q' \in \mathcal{L}(Q)$, $Q \equiv Q'$ if and only if $Q(demo_{Q'}) = Q'(demo_{Q'})$.* ∎

Our final example illustrates the construction of demo databases from exemplars and shows how demo databases circumvent the limitations of exemplars that apply to set semantics.

Example 5 *Consider the target query Q_d:*

```
SELECT
DISTINCT name
FROM     PATIENT p, DIAGNOSIS d
WHERE    name='Smith' AND p.pid='NHI003'
         AND d.pid='NHI003' AND condition='HIV';
```

Our construction for Q would yield representative tuples:

p.pid	*p.name*	*d.pid*	*d.condition*
NHI003	*Smith*	*NHI003*	*HIV*
NHI003	*Smith*	*NHI003*	*Fever*
NHI003	*Smith*	*QGP700*	*HIV*
NHI003	*Ryan*	*NHI003*	*HIV*
KGL050	*Smith*	*NHI003*	*HIV*

and the following $\mathcal{L}(Q)$-exemplar:

PATIENT	
pid	*name*
NHI003	*Smith*
NHI003	*Ryan*
KGL050	*Smith*

DIAGNOSIS	
pid	*condition*
NHI003	*HIV*
NHI003	*Fever*
QGP700	*HIV*

For the user query Q_d^1:

```
SELECT
DISTINCT name
FROM     PATIENT p, DIAGNOSIS d
WHERE    name='Smith' AND p.pid='NHI003'
         AND d.pid='NHI003';
```

we would obtain the demo database $demo_{Q_d^1}$ where

PATIENT	
pid	*pname*
NHI003	*Smith*

DIAGNOSIS	
pid	*condition*
NHI003	*Fever*

since $t_1 = (NHI003, Smith, NHI003, Fever)$ represents the minimal proper container Q^1 of Q. Here, Q_d^1 returns {Smith} while Q_d returns an empty answer when evaluated on $demo_{Q_d^1}$. Note that Q^1 returns {Smith,Smith} while Q returns {Smith} when evaluated on db_Q. For the user query Q_d^2:

```
SELECT
DISTINCT name
FROM     PATIENT p, DIAGNOSIS d
WHERE    p.pid='NHI003' AND d.pid='NHI003'
         AND condition='HIV';
```

we would obtain the demo database $demo_{Q_d^2}$ where

PATIENT	
pid	*pname*
NHI003	*Ryan*

DIAGNOSIS	
pid	*condition*
NHI003	*HIV*

since $t_2 = (NHI003, Ryan, NHI003, HIV)$ *represents the minimal proper container* Q^2 *of* Q*. Here,* Q^2_d *returns* $\{Ryan\}$ *while* Q_d *returns an empty answer when evaluated on* $demo_{Q^2_d}$*. Note that* Q^2 *returns* $\{Smith,Ryan\}$ *while* Q *returns* $\{Smith\}$ *when evaluated on* db_Q*. For a final user query* Q'_d*:*

```
SELECT
DISTINCT name
FROM     PATIENT p, DIAGNOSIS d
WHERE    name=‘Smith’ AND p.pid=‘NHI003’
         AND p.pid=d.pid AND condition=‘HIV’;
```

we would obtain the demo database $demo_{Q'_d}$ *where*

PATIENT	
pid	*pname*
NHI003	*Smith*

DIAGNOSIS	
pid	*condition*
NHI003	*HIV*

since $t_0 = (NHI003, Smith, NHI003, HIV)$ *is a tuple that represents* Q*. Since* Q *and* Q' *are equivalent, they produce matching answers on any database, including* $\{Smith\}$ *when evaluated on* $demo_{Q'_d}$*.*

9 Conclusion

This article investigated the concept of exemplars for conjunctive queries without self-joins. A database db_Q is an $\mathcal{L}(Q)$-exemplar for such a query Q if and only if for every query $Q' \in \mathcal{L}(Q)$, Q' and Q have matching answers on every database if and only if Q' and Q have matching answers on db_Q. Therefore, having an $\mathcal{L}(Q)$-exemplar db_Q for Q reduces the query equivalence problem for Q and any $Q' \in \mathcal{L}(Q)$ to a simple evaluation of Q and Q' on db_Q. As the first main contribution a construction of an $\mathcal{L}(Q)$-exemplar db_Q was given for bag semantics, the default semantics in SQL. Here, $\mathcal{L}(Q)$ consists of all conjunctive queries Q' that have the same `SELECT` and `FROM` clause as Q, respectively, and every constant in Q' already occurs in Q. In contrast, $\mathcal{L}(Q)$-exemplars do not always exist under set semantics. For set semantics the new concept of a demo database shows how to use $\mathcal{L}(Q)$-exemplars to generate for each query $Q' \in \mathcal{L}(Q)$ a database $demo_{Q'}$ of singleton relations such that Q and Q' have matching answers on $demo_{Q'}$ if and only if they have matching answers on every database. The second main contribution is an implementation of the exemplar construction in form of the graphical user interface LECQTER. Here, trainers can construct exemplars for a given target query Q, and trainees can submit queries from $\mathcal{L}(Q)$ to find out whether their query is semantically correct, that is, equivalent to Q. The benefits for learning conjunctive queries using LECQTER have been emphasized throughout the article. The third main contribution is a detailed performance analysis of LECQTER in terms of the size of the exemplars it constructs and the time required for their construction. Exemplars may contain exponentially many tuples in the number of participating attributes, yet they are able to separate exponentially many non-equivalent queries in that size. It was found that the worst cases, when the size and time are exponential in that of the input, are unlikely to occur, yet their construction is still efficient for learning how to write sound conjunctive queries. The general problem of deciding whether a given database is an exemplar for a given conjunctive query appears to be difficult - and its time complexity is not obvious. Therefore, our construction of exemplars is particularly useful. Indeed, trainers can construct exemplars and populate them with real-world values.

10 Future Work

Several directions for future work arise from the research. Theoretically speaking, the most prominent related problem is to identify the exact time complexity of deciding the containment problem for conjunctive SQL queries, or in other words, for conjunctive queries under bag semantics. Despite several attempts, the problem has remained open for the last twenty years [4, 9, 10, 13, 14]. Similarly, it would be interesting to study the time complexity of deciding whether a given database is an exemplar for a given conjunctive query under bag semantics. It would also be valuable to increase the expressivity of the query language for which exemplars do exist, and to appropriately extend our techniques regarding their construction. In fact, we have shown that our construction requires just a simple extension to cover the case where arbitrary constants can occur in the user query. In that case tuple pairs are required to represent queries, as a query $Q' \subsetneq Q$ may now select the tuple t_0 that previously represented Q by itself. For example, by adding the equality `p.pid='p1'` to the `WHERE` clause of our target query Q from Example 1 we obtain a query $Q' \subsetneq Q$. The tuple t_0 from Example 4 that represented Q so far by itself now also satisfies $\varphi_{Q'}$. However, by adding another tuple $\bar{t}_0$ that represents Q, but uses different values from those in t_0 wherever possible, it can be ensured that t_0 and $\bar{t}_0$ both satisfy Q' if and only if $Q \subseteq Q'$ holds. In our example we may add the tuple $\bar{t}_0$=(c7,Jack,p0,scanner,c7,p0,3/9/2014,86). The resulting theory is more involved than that presented in the current article, and warrants follow-up research. This holds even more so in the case of self-joins.

Another challenge is to investigate the problems of this article in the context of database constraints that naturally put restrictions on the patterns that can occur in databases [11]. Here the problem changes significantly as equality-generating constraints such as keys or functional dependencies may enforce some equalities that result in the equivalence of queries that are not equivalent in the absence of the constraints. The same applies to tuple-generating constraints such as foreign keys and multivalued dependencies.

Practically speaking, the next step would be to conduct empirical investigations regarding the usability of Lecqter within real teaching environments. Therefore, it would be interesting to observe groups of students in learning conjunctive SQL queries with and without the support of Lecqter. This should provide insight into the extent to which the tool can help trainees learn, but also the extent to which the tool helps trainers save time that they can use instead for other important tasks such as discussions.

Another sensible next step would be to exploit the application for the automated marking of conjunctive SQL queries in database courses, and see whether it effectively reduces the complexity of marking and the amount of time that trainers spend on marking. In this direction it would be interesting to identify the complexity of queries that offer the right amount of flexibility to assign appropriate marks.

References

[1] M. Andreesen. Why software is eating the world. `http://online.wsj.com/news/articles/SB10001424053111903480904576512250915629460`, 2011.

[2] K. Bache and M. Lichman. UCI machine learning repository, 2013.

[3] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[4] S. Chaudhuri and M. Y. Vardi. Optimization of real conjunctive queries. In *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 59–70. ACM, 1993.

[5] S. Doherty. The future of enterprise data: RDBMS will be there. `http://insights.wired.com/profiles/blogs/the-future-of-enterprise-data\#axzz2owCB8FFn`, 2013.

[6] R. Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.

[7] Free Software Foundation Inc. SQLTUTOR. `http://sqltutor.fsv.cvut.cz/cgi-bin/sqltutor`.

[8] S. Hartmann, M. Kirchberg, and S. Link. Design by example for SQL table definitions with functional dependencies. *VLDB J.*, 21(1):121–144, 2012.

[9] Y. E. Ioannidis and R. Ramakrishnan. Containment of conjunctive queries: Beyond relations as sets. *ACM Trans. Database Syst.*, 20(3):288–324, 1995.

[10] T. S. Jayram, P. G. Kolaitis, and E. Vee. The containment problem for real conjunctive queries with inequalities. In *Proceedings of the Twenty-Fifth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 80–89, 2006.

[11] D. S. Johnson and A. C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28(1):167–189, 1984.

[12] J. Kay, P. Reimann, E. Diebold, and B. Kummerfeld. MOOCs: So many learners, so much potential .. *IEEE Intelligent Systems*, 28(3):70–77, 2013.

[13] P. G. Kolaitis. The query containment problem: Set semantics vs. bag semantics. In *Proceedings of AMW*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[14] S. Kopparty and B. Rossman. The homomorphism domination exponent. *Eur. J. Comb.*, 32(7):1097–1114, 2011.

[15] H. Mannila and K.-J. Räihä. Design by example: An application of Armstrong relations. *J. Comput. Syst. Sci.*, 33(2):126–141, 1986.

[16] H. Mannila and K.-J. Räihä. Test data for relational queries. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 217–223, 1986.

[17] H. Mannila and K.-J. Räihä. Automatic generation of test data for relational queries. *J. Comput. Syst. Sci.*, 38(2):240–258, 1989.

[18] Mediawiki. SQLZoo. `http://sqlzoo.net/wiki/SELECT_basics`.

[19] S. Moiseenko, O. Lysenko, D. Valuev, V. Dolgopolov, E. Krasovskij, P. Kurochkin, A. Maistrenko, and V. Kalinkin. SQLEXERCISE. `http://www.sql-ex.ru/exercises.php`.

[20] M. Orlov. Efficient generation of set partitions. Technical report, University of Ulm, 2002.

[21] J. D. Ullman. Gradiance on-line accelerated learning. In *ACSC*, pages 3–6, 2005.

[22] M. Y. Vardi. Will MOOCs destroy academia? *Commun. ACM*, 55(11):5, 2012.