Semi-automatic SQL Debugging and Fixing to solve the Missing-Answers Problem

Katerina Tzompanaki

Semi-automatic SQL Debugging and Fixing to solve the Missing-Answers Problem

Katerina Tzompanaki
Ph.D. Student
supervised by Nicole Bidoit and Melanie Herschel
University Paris Sud & Inria Saclay, Orsay, France
tzompana@lri.fr

ABSTRACT

Asking questions is the driving force for scientific progress. But as important as it is to ask questions, so important is to be able to understand the obtained answers. In this way, we are able to verify the sanity of the question itself and if necessary refine it. In the same spirit, in this PhD we focus on SQL queries over relational databases with the aim to (1) pinpoint the reasons on the SQL query that led to missing-answers (tuples that were expected but not obtained in the query result), (2) fix the SQL query so that it best fits the user's expectations, and (3) do both efficiently so as to be of practical use.

1. CONTEXT AND MOTIVATION

One frequent task that developers perform when processing large amounts of data is to define complex data transformations, commonly in the form of queries. However, unexpected data in the output of a query can trigger a whole new series of tedious and error-prone query debugging and query rewriting tasks. Take for example a developer, who may miss some tuples from the result of an SQL query and/or receive undesired ones. Consequently, she might ask the following questions: *"Why did this happen?"* and *"How can this unexpected and undesired behavior be fixed?"*. To answer them, the developer traditionally first traces the origins of the expected and unexpected result in the source database to then further analyze how the query manipulates these data. This allows to obtain a better understanding of the reasons that prevented the correct answers to appear in the result. She then continues by altering the query or even the data acquisition process before testing again in the new setting. If the result is not yet satisfactory, this whole manual process has to be iterated, possibly many times.

The procedure described above reminds us of 'classical' debuggers and development tools put at programmers disposal for procedural programming languages. However, there is yet no system that can act as debugger for declarative queries and data transformations. First steps towards debugging include [2, 20]. However, especially as queries become more complex, the developer still has to face the challenge of manually searching and identifying the problematic parts, and no alternatives are proposed to fix the query once the erroneous clauses have been identified.

The overall goal of the Nautilus project [23] is to provide semi-automatic algorithms and tools for data transformation analysis, fixing, and testing. The PhD thesis presented in this paper is set in this context and aims at devising algorithms to support SQL debugging and fixing when facing one particular type of problem, i.e., the lack of some expected data in the query result. We refer to this problem as the *missing-answers* problem.

In presenting our work, firstly Sec. 2 reviews related work. Sec. 3 summarizes the research questions we address and highlights our scientific contributions. In Sec. 4 and Sec. 5, we describe our contributions so far and outline future work, respectively. Finally, we conclude in Sec. 6.

2. RELATED WORK

In this section we are surveying the works that are relevant to the developer questions introduced in Sec. 1: why are there (or not) certain tuples in some transformation result (Sec. 2.1) and how can we alter the transformation to obtain the desired results (Sec. 2.2).

2.1 Provenance & Transformation Debugging

General discussion. The *data provenance* field, which started with the introduction of *data lineage* in relational databases, counts more than one decade of study [38]. As surveyed in [9], *data provenance* research focuses on explaining data present in a query result and may be categorized in three forms based on *where* [7] the data were copied from, *why* [7, 10] a query answer was produced (i.e., based on what source data), and *how* [17] data were manipulated by the query to produce the result data in question. As discussed in [22], approaches explaining missing-answers may focus on the query and point out how source data were lost [8], or they focus on the source data and explain why these data cannot yield the desired output [25]. We discuss these approaches in more detail below, as these are the most relevant w.r.t. this thesis. Note that our work is interested in *fine-grained* provenance at the level of individual data items (e.g., tuples) as opposed to *coarse-grained* provenance that remains at schema or manipulation type signature level, as is commonly the case in workflow provenance.

Further approaches useful for query debugging include methods for weighting and ranking the possibly large provenance, e.g., based on causality and responsibility [31] or methods to automatically generate test data given a query and a desired output [6, 33]. The latter are particularly valuable for instance when the source data are not accessible or incomplete. Further support in verifying transformation behavior and semantics can be obtained through sub-query result inspection [20], visualization [14], or tools that simplify the specification of complex data transformations [29, 32].

Proceedings of the VLDB 2014 PhD Workshop.

Whereas the methods mentioned above focus on relational data and queries, other approaches have considered more complex data transformations where in the worst case, individual manipulations use black-box functions. Clearly, data transformation debugging and fixing is essential in this context, and first solutions to incorporate data provenance in scientific workflows [3] or Map-Reduce workflows [34] have been proposed. We further observe that fundamental ideas underlying data provenance techniques are not limited to relational data, as it has also been considered for Datalog [18], logic programming [13], XML [15], and SPARQL [12].

Explaining Missing-Answers. Related algorithms can be divided into returning instance-based, query-based, or hybrid explanations.

Algorithms returning instance-based explanations [24, 25] look for problems in the data and output sets of tuples that, if added to the database would produce the missing-answers in the query result. Their main difference lies in the class of missing-answers and SQL queries they can handle. [35] approaches the problem of explaining numerical queries by returning the top-k tuples that are more responsible for the observed result, in a more formal way.

Query-based explanations place the responsibility of the missing-answers not on the source data, but on the transformation (or query). As such, query-based explanation algorithms return a set of transformation operators that have pruned the desired answers. Initially, this solution was proposed for general workflows represented as DAGs of manipulations [8], while it can further be specialized to relational queries. In this case, the DAG representation of a query is an algebraic tree. This method has several shortcomings that we addressed in [5], to propose a more effective and efficient algorithm. While both these approaches consider one particular instance of an algebraic query tree, their result may differ for equivalent algebraic query trees, an issue that we recently addressed in [4]. We summarize both our contributions in Sec. 3.

There are cases where pure query-based or instance-based explanations cannot provide a helpful indication, as they require a combination of both kinds of reasons for the missing-answers. In such cases, hybrid explanations [22] suggest both the tuples to add to the source database and the operators of the algebraic query tree that pruned the desired answer, after applying the suggested hypothetical source database changes.

2.2 Query Modification and Fixing

Once the reasons for unexpected transformation results have been identified, a developer wonders how to leverage this knowledge to obtain the needed results. Often, this requires changing the data transformation, in which case the developer has to manipulate the culprit operators returned as query-based explanations. This task becomes time-consuming when we consider the number of different options of changing a transformation. For instance, given that a particular join is returned as query-based explanation, should she change the join condition, replace it with some outer join, or should it actually be a union? In this section, we review some work where transformations are automatically adjusted to changing requirements in general before we focus on modifying queries that make missing-answers appear in the result.

General discussion. A number of works about query rewriting concern schema mappings adaptation. For example, [11] proposes a solution in cases where the source schema evolves but the target schema and result are invariant. Similarly, [39] describes an approach to adapt schema mappings when the both source and target schemas evolve. To check the correctness of a transformation, [36] generates test databases that kill mutants of a specific query and can be further used to identify inequivalent query rewritings. [16] rewrites the query in order to annotate the result tuples with provenance information, in the form of extra attributes.

Unlike the aforementioned rewriting requirements, our goal is to rewrite a query keeping the schemas the same but not the target (result) data. This requirement derives from the fact that the desired result should be the original one plus the missing-answers.

Query Modification for Missing-Answers. [37] considers a select-project-join-aggregation (SPJA) query and a complex combination of missing tuples and returns a query similar to the original that is capable of including the missing-answer to the result set with minimal side effects, i.e., by adding to the result set as few extra tuples as possible (ideally only the missing-answers). Given a reverse skyline query and a missing data point, [27] proposes modifications both on the query and the missing data point. [21] addresses the same problem but for top-k queries by adapting either k and/or the coefficients of a preference query so as to make the missing-answer appear in the result of the re-written top-k query.

3. THESIS CONTRIBUTIONS

Given the research context and the discussion of related work, we now clearly state the problems, scope, and contributions that we envision for this PhD thesis. The problem statements include some restrictions on the scope and some deliberate imprecisions, which we further clarify in the rest of the discussion.

PROBLEM STATEMENT 3.1 (QUERY ANALYSIS.). *Given an SQL query Q, a database instance $\mathcal{I}$, and a set of tuples T disjoint from the result $Q(\mathcal{I})$, how can we effectively and efficiently generate useful query-based explanations for the missing-answers defined by T?*

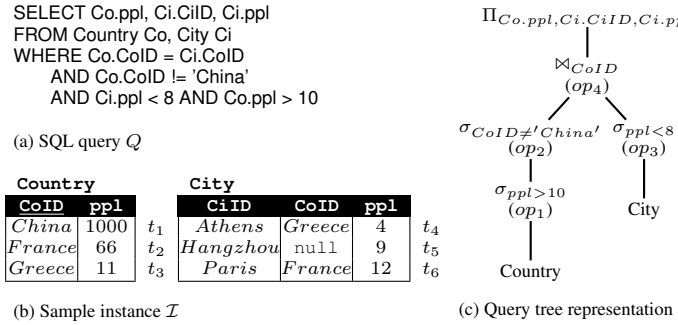
PROBLEM STATEMENT 3.2 (QUERY FIXING.). *With the same hypothesis as above, given the query-based explanations $\mathcal{X}$, how can we efficiently compute a set of useful queries $\mathcal{Q}'$ such that $\forall Q' \in \mathcal{Q}', Q'(\mathcal{I}) \approx Q(\mathcal{I}) \cup T$?*

SQL queries. SQL has been accepted for a long time as the standard language to manipulate relational data, and thus is one of the most popular query languages [19]. So, studying query debugging and fixing for SQL queries appears to us as a very natural and practically relevant choice.

Query-based explanations. We consider that the content of the database is trusted and not subject to change, so reasons for the missing-answers can only be placed on the constraints imposed by query operators. This information guides a developer to subsequently fix the query. But even if the source data were modifiable, existing work [8, 22] indicate that the numerous instance-based explanations alone are overwhelming and quite costly to compute.

Furthermore, it remains an open question how to 'package' the query operators that are part of query-based explanations to make them most useful to the developer. Is it better to return only one candidate and if so, which one ([8] opted for this approach) or is it better to return a (ranked) list of all potential responsible operators? Moreover, should data examples be returned along with the operators? This thesis defines and analyzes some alternative solutions, devise algorithms computing these and also evaluates them experimentally. First results are published in [4, 5].

Query fixing. The numerous options for fixing a query may be frustrating for a developer, even if she knows in what query conditions the problem lies. (Semi-)automatically generating appropriate query modifications can relieve the developer from this effort. An interesting and fundamental preliminary question is which query modifications can be considered as appropriate. Ideally, each rewritten query $Q' \in \mathcal{Q}'$ will return the result of the original query



(a) SQL query Q

(b) Sample instance $\mathcal{I}$

(c) Query tree representation

Figure 1: Sample query (a), instance (b), and query tree (c)

Q plus the missing-answers T, i.e., $Q'(\mathcal{I})=Q(\mathcal{I})\cup T$, and thus avoid side-effects. However, finding such a solution is not always possible or the problem may be too complex to be solved efficiently. Therefore, this thesis aims at devising approximate solutions (hence the use of $\approx$ in the problem statement). Further pruning and ranking of the refined queries can be used to provide only useful solutions, useful meaning relevant to the initial intentions.

Formalization. Even though explaining and recovering missing-answers have recently been considered, there is surprisingly little formalization of the problem and the principles underlying algorithms. In this thesis, we put a special emphasis on clearly formalizing the general problem and our solutions, as showcased in our publications so far [4, 5].

Efficient Solutions. Calculating explanations of missing-answers and providing query modifications can be very complex depending on the size of the database and the query complexity. To be of any use to a developer especially in an interactive setting, we ought to produce useful explanations and fixes in an efficient manner. In this thesis, we will investigate efficiency of our algorithms both through classical complexity analysis and experimental evaluation. So far, no existing solution addressing the missing-answers problem in SQL query results has explicitly considered efficiency.

Implementation. An integral part of this thesis is to implement and evaluate the designed algorithms, with a special focus on making the algorithms output easily consumable by developers. This implementation will extend the Nautilus Analyzer that was previously developed in our team [23].

4. QUERY-BASED EXPLANATIONS

So far, we have addressed some of the aforementioned research questions in two works proposing the *NedExplain* [5] and the *Ted* [4] algorithms. The cornerstones of both algorithms are the same; they both compute query-based explanations for missing-answers to address Problem Statement 3.1. However, there are subtle differences w.r.t. their application scope and principles. The discussion first highlights the main concepts underlying both algorithms before we delve into selected details for each one. To illustrate the presentation, we use the sample data and query shown in Fig. 1. We show both an SQL and a query tree representation of the query, the latter providing labels for the individual query operators. Similarly, the tuples in the sample instance are labelled for future reference. The obtained answer is $(11, Athens, 4)$.

4.1 Common Concepts

Both algorithms allow to specify the set of missing-answers in form of a Why-Not question, as illustrated in Example 4.1:

EXAMPLE 4.1. *Considering the data and query of Fig. 1, let us assume that a developer wonders why she did not get the city named $Hangzhou$ in some country with a population (ppl) greater or equal to 1000. This Why-Not question can be formally described by a conditional tuple t_c=$((Co.ppl{:}x_1, Ci.CiID{:}x_2), x_1{\geq}1000 \wedge x_2{=}Hangzhou)$, or more briefly t_c=$((Co.ppl{:}x_1, Ci.CiID{:}Hangzhou), x_1{\geq}1000)$. Assume now that the developer wonders why she did not get any result tuple having the country population lower than the city population. This Why-Not question is expressed by t'_c=$((Co.ppl{:}x_1, Ci.ppl{:}x_2), (x_1{<}x_2))$.*

Having an intuition on how the Why-Not question is expressed, we can now proceed to the formal definition:

DEFINITION 4.1 (WHY-NOT QUESTION). *A Why-Not question w.r.t. a query Q is a predicate $\mathcal{P}$ over Q's target type $\mathcal{T}_Q$, where $\mathcal{P}=\bigvee_{i=1}^{n} t_c^i$, with t_c^i being a conditional tuple [26] s.t. type$(t_c^i)\subseteq\mathcal{T}_Q$.*

Both algorithms rely on this definition. NedExplain considers Q to be a SPJA query or a union thereof, and relies on a logical query tree representation of the query for further processing. Additionally, a conditional tuple may only compare attributes with constant values or with attributes of the same relation. On the other hand, Ted considers a conjunctive query and relies on its tableau representation [1]. Furthermore, in Ted, conditional tuples may also compare attributes of different relations. As a result, in the Example 4.1 the first conditional tuple t_c can be treated by both algorithms, while t'_c that compares attributes from different relations, can be handled only by Ted.

Having the missing-answers modelled by the Why-Not question, we gather all the source tuples that are relevant to produce these missing-answers. To identify the relevant source tuples, we use the notion of *compatibility* w.r.t. a conditional tuple t_c. As the two algorithms differ in the expressiveness of the conditional tuple, they consequently differ in their respective notions of compatibility. NedExplain does not allow individual constraints to span over more than one relation thus, it is sufficient to identify compatible tuples per relation. On the contrary, Ted allows for constraints involving more than one relation and thus, identifies combinations of tuples from different source relations. Each such tuple combination forms what we call a *concatenated compatible tuple* (*cc-tuple*). Indeed, a compatible tuple is a special case of a cc-tuple.

EXAMPLE 4.2. *For the conditional tuple t_c, we obtain the compatible tuples t_1 and t_5. Indeed, t_5 satisfies the constraint on $City$ (as it has $CiID{=}Hangzhou$) whereas t_1 satisfies the constraints on $Country$ ($Co.ppl{\geq}1000$). Concerning t'_c, we have to concatenate tuples from the different relations to check whether the condition is satisfied. For example, the tuple t_3 can be considered compatible w.r.t. t'_c only in association with tuple t_6 (Paris has a greater population than Greece). We denote the resulting cc-tuple as τ=(t_3t_6).*

The fundamental concept that comes next is *pickyness*, a property that defines those query parts that pruned the compatible source data. As already mentioned, the query Q considered in the two algorithms is not represented in the same way as the one uses a query tree and the other a minimal tableau. As a consequence, NedExplain identifies picky subqueries, whereas Ted more generally returns picky query conditions.

EXAMPLE 4.3. *For the Why-Not question involving t_c, NedExplain identifies two picky subqueries. Referring to Fig. 1, these*

correspond to the subqueries rooted at op_2 and op_3. Indeed, the valuation for ppl in the compatible tuple t_5 does not satisfy the selection condition of op_3. Similarly, the attribute values of t_1 contradict the condition of the selection op_2.

Note that NedExplain does not return culprit subqueries rooted higher up in the query tree as all compatible tuples have been lost once tracing them all the way up to op_2 and op_3. This is a consequence of NedExplain considering one particular instance of a query tree and that the result is w.r.t. this query tree. Consequently, equivalent query trees may yield different sets of picky operators when using NedExplain. For example, if all selections are pushed above the join, only the join would be identified as picky, as it will be responsible for pruning both compatible tuples t_1 and t_5. Contrary to that, Ted returns all conditions (that are linked to query operators) that prune compatible data (cc-tuples in general) independently of a query tree representation.

EXAMPLE 4.4. *For the Why-Not question t_c defines, Ted identifies as picky the conditions $Co.CoID \neq `China'$, $Ci.ppl < 8$, and $Co.CoID = Ci.CoID$ as the values of the cc-tuple $(t_1 t_5)$ do not satisfy any of these conditions. Thus, Ted identifies the picky operators op_2, op_3, op_4.*

As a final step, both algorithms return query-based explanations, later called Why-Not answers, for the missing-answers captured by the Why-Not question. NedExplain models its Why-Not answer as a set of picky subqueries linked to the compatible tuples that were excluded at that point. So, not only is the user informed about the subqueries that are to blame but also gets a guidance on how to fix them, by knowing the values of the attributes that do not agree with the picky subqueries' constraints. Ted models the Why-Not answer as a polynomial of picky operators. Essentially, an addend $k * op_1 * \cdots * op_n$ provides a query-based explanation (the operators $op_1, \ldots, op_n$). Moreover, the coefficient k gives an estimation of the amount of different ways in which the missing-answer could be generated if the given picky operator combination was passing.

EXAMPLE 4.5. *Continuing our example using t_c, NedExplain returns the Why-Not answer $\{(t_1, op_2), (t_5, op_3)\}$ while Ted returns $op_2 * op_3 * op_4$. Assuming that table $City$ did contain another tuple $t'_5 = (Hangzhou, China, 9)$, NedExplain would return $\{(t_1, op_2), (t_5, op_3), (t'_5, op_3)\}$ whereas Ted would return $op_2 * op_3 * op_4 + op_2 * op_3$, i.e., there is one addend for the cc-tuple $(t_1 t_5)$ and another one for the cc-tuple $(t_1 t'_5)$.*

Overall, comparing the two algorithms, NedExplain can handle a wider class of queries and is more efficient than Ted. On the other side, Ted's main strengths are that it can handle a wider class of Why-Not questions and that it provides a complete set of query-based explanations w.r.t. different query tree re-orderings as opposed to NedExplain. However, as we will see below, this comes at the price of a high worst-case complexity.

We now discuss the specificities of both algorithms in more detail. Due to space limitations most formal definitions are omitted and interested readers can refer to the detailed papers [4, 5].

4.2 NedExplain

Given an algebraic query tree and after identifying compatible tuples as described in the previous section, the core of the NedExplain algorithm [5] traverses the query tree in a bottom-up way. When at some node (i.e., query operator at the root of a subquery) it loses track of compatible tuples, meaning they can be found in the operator input but not in the output, it marks the subquery rooted at the operator as a *picky* subquery. Example 4.6 illustrates this process by executing the algorithm on the running example and therefore describes in detail how we obtained the result in Example 4.3.

EXAMPLE 4.6. *We are going to trace the compatible tuples t_1 from $Country$ and t_5 from $City$ on the query tree in Fig. 1(c). Let us start with the subquery rooted at op_1. After executing it on the $Country$ instance (that contains the compatible tuple t_1), we observe that there is a tuple that is generated using only compatible tuples (i.e., the tuple t_1). We say that t_1 has a* valid successor *in the output of op_1. Then we move to op_2. In the output of op_2 we do not find any valid successors of t_1, so we mark op_2 as a picky subquery for t_1. Similarly, in the input of op_3 we have the compatible tuple t_5, however we do not obtain a valid successor of it in the output. Thus, op_3 is picky for t_5. Now, in the input of op_4 there are not any successors of compatible tuples and so we stop the procedure of searching for picky subqueries.*

The Why-Not answer returned by NedExplain includes the picky subqueries identified in Q and can take three forms. For brevity, here, we focus on the most informative one, i.e., the *detailed Why-Not answer*, previously illustrated in Example 4.5. Assuming a Why-Not question with a single conditional tuple, we define this Why-Not answer as formalized in Definition 4.2 below. When the Why-Not question contains more than one conditional tuple, the Why-Not answer is the union of the Why-Not answer obtained for each conditional tuple. Note that this definition distinguishes between direct and indirect compatible tuples (sets Dir^{t_c} and $InDir^{t_c}$, respectively). Intuitively, the direct compatible tuples are those that have attributes constrained in the Why-Not question. Thus they are of direct interest for the user and they can be used in conjunction with the picky subqueries for query re-writing purposes. The indirect compatible tuples are not linked to the Why-Not question and correspond to necessary data for the query.

DEFINITION 4.2 (DETAILED WHY-NOT ANSWER). *The detailed Why-Not answer of t_c w.r.t. Q and $\mathcal{I}$, is defined as*

$$\bigcup_{t_{\mathcal{I}} \in Dir^{t_c}} \{(t_{\mathcal{I}}, Q') \mid \begin{array}{l} Q' \text{ subquery of } Q \text{ and} \\ Q' \text{ picky w.r.t. } Dir^{t_c} \cup InDir^{t_c} \text{ and } t_{\mathcal{I}}\} \end{array} \text{ where}$$

$Dir^{t_c} \cup InDir^{t_c}$ is the set of all compatible tuples in $\mathcal{I}$.

The algorithm producing the detailed Why-Not answer is discussed in detail in [5]. Its worst case time complexity is in $O(|Q|(L + Out))$.$|Q|$ denotes the size of Q, Out is the worst case size (in terms of number of tuples) of a subquery's output, and L is the height of the query tree.

As previously highlighted in Sec. 4.1, the picky subqueries returned may vary for different equivalent query trees of the same query Q. Avoiding this behavior was our main motivation when designing the Ted algorithm.

4.3 The Ted Algorithm

Ted algorithm [4] is divided in two main phases: (1) computing the set of all cc-tuples and (2) determining the Why-Not answer.

Computing the set of all cc-tuples. As seen in Sec. 4.1, Ted allows a more general class of Why-Not questions with comparisons between attributes from different relations, which entails the necessity of defining cc-tuples. To compute the set of cc-tuples, we use a *compatibility* tableau T_{t_c} that is built using the input query schema $\mathcal{S}_Q$ (in this case *Country* and *City*) and the condition of t_c.

EXAMPLE 4.7. *Tab.1 illustrates the compatibility tableau for the query of Fig. 1 and the conditional tuple $t'_c = ((Co.ppl{:}x_1, Ci.ppl{:}x_2), (x_1 < x_2))$ introduced in Ex. 4.1 (the tableau omits the summary for brevity). There is a straightforward mapping from variables in the conditional tuple to variables in T_{t_c}, e.g., x_1 and x_2 in t'_c map to x_2 and x_5 in the tableau, respectively.*

Table 1: Tableau T_{t_c} for sample Why-Not question t'_c

	Co.CID	Co.ppl	Ci.CiID	Ci.CoID	Ci.ppl	cond
Country Co	x_1	x_2				$x_2 < x_5$
City Ci			x_3	x_4	x_5	$x_2 < x_5$

Even though in Example 4.2, we only show one cc-tuple, in general, we obtain a set of cc-tuples w.r.t. T_{t_c}, which we denote $CCT(T_{t_c}, \mathcal{I})$. Each $\tau \in CCT(T_{t_c}, \mathcal{I})$ would have contributed the missing-answer, *if* it succeeded to satisfy *all* query conditions.

To efficiently compute $CCT(T_{t_c}, \mathcal{I})$, we create a *valid partitioning* of T_{t_c} that groups together those rows of the tableau that share some variables in their conditions. Then, for each partition, we first join the involved relations (one row in the tableau stands for a relation) based on their join conditions (conditions with variables from two rows) and apply the remaining selection conditions. Thus, we obtain a set of cc-tuples for each partition.

We then use the following lemma [4] to compute the full set $CCT(T_{t_c}, \mathcal{I})$, by performing the cross product on the sets of cc-tuples of each partition.

LEMMA 4.1. *Let $Part = \{Part_1, \ldots, Part_k\}$ be the valid partitioning of T_{t_c} and $\mathcal{I}$ be a well-typed database instance. Then,* $CCT(T_{t_c}, \mathcal{I}) = \bigtimes_{Part_i \in Part} CCT(T_{Part_i}, \mathcal{I}_{|Part_i})$.

Computing the Why-Not answer. Essentially, the Why-Not answer includes, for each $\tau \in CCT(T_{t_c}, \mathcal{I})$, *all* query conditions that 'picked' τ out of the result. To compute the picky conditions and thus the picky operators for each $\tau \in CCT(T_{t_c}, \mathcal{I})$ we use another tableau T_τ, as illustrated in Tab. 2. This tableau has two columns that are used for conditions: (1) in $cond_\tau$ we keep the conditions on the attributes of each row induced by a given τ (here $(t_3 t_6)$) and (2) in $cond_Q$ we keep the conditions on the attributes of each row imposed by Q (here the selection and join conditions of op_1, op_2, op_3, and op_4). When some condition $cond_p \in cond_Q$ contradicts some constraint in $cond_\tau$, then $cond_p$ is picky for τ.

EXAMPLE 4.8. *In the Country row of Tab. 2, the conditions on the variables x_1 and x_2 in the two condition columns, agree. However, the query condition $x_1 = x_4$ is not satisfied by the valuation of τ, as $Greece \neq France$. So, $x_1 = x_4$ and hence its associated query operator op_4 is picky for τ. Similarly, we identify that op_3 (associated with the query condition $x_5 < 8$) is picky for τ and thus the set containing the picky operators w.r.t. τ is $PO_\tau = \{op_3, op_4\}$. To model that both operators are problematic w.r.t. τ, we introduce the product $op_3 * op_4$ associated to τ.*

As each cc-tuple represents one alternative to obtain the missing-answers, we determine one product for each τ in $CCT(T_{t_c}, \mathcal{I})$. We then sum up these products and obtain a polynomial that explains *how* the operators in the query are picky w.r.t. t_c. "How" here means in which combination and in how many ways, and also acknowledges the fact that this representation is similar in spirit to the provenance polynomials used for how-provenance [17]. Formally, we define the Why-Not answer that Ted returns as follows:

DEFINITION 4.3. *(Why-Not answer w.r.t. t_c) Given a query Q over a database schema $\mathcal{S}_Q$, the instance $\mathcal{I}$ over $\mathcal{S}_Q$, and the compatibility tableau T_{t_c} associated with the Why-Not question t_c, we define the Why-Not answer w.r.t. t_c as* $\sum_{\tau \in CCT(T_{t_c}, \mathcal{I})} \prod_{op \in PO_\tau} op$.

Assuming that the number of tuples $|\mathcal{I}_{|R}|$ of a relation R is typically much larger than the size of the schema or query, the worst case complexity of Ted is in $O(N^k)$, where k is the number of relations and N the maximum size of a relation instance. Clearly, the complexity of the algorithm is prohibitive for large databases and so our first priority is to address efficiency.

Table 2: Tableau T_τ for cc-tuple $\tau = (t_3 t_6)$

	Co.CID	Co.ppl	Ci.CiID	Ci.CoID	Ci.ppl	$cond_\tau$	$cond_Q$
Country Co	x_1	x_2				$x_1 = Greece \wedge$ $x_2 = 11$	$x_1 \neq China \wedge$ $x_2 > 10 \wedge x_1 = x_4$
City Ci			x_3	x_4	x_5	$x_3 = Paris \wedge$ $x_4 = France \wedge$ $x_5 = 12$	$x_5 < 8 \wedge$ $x_1 = x_4$

4.4 Implementation and Evaluation

We implemented both NedExplain and Ted using Java and evaluated their performance in terms of effectiveness and efficiency on several benchmark scenarios. We also implemented the algorithm presented in [8], the only other algorithm returning query-based explanations for SQL queries (our implementation did benefit from source code kindly provided by the authors of [8]). A detailed discussion of the experimental setup and results can be found in [5, 4]. Briefly, *NedExplain* produces better quality answers than [8], by quality meaning that it captures also picky subqueries that [8] is not able to capture. Additionally, *NedExplain* outperforms [8] in terms of execution time (even though worst-case complexities are comparable) [5]. However, even though the picky operators calculated by *NedExplain* and [8] vary for different choices of query trees, they are always included in the Why-Not answer by Ted.

5. FUTURE DIRECTIONS

Based on our work so far, we now briefly discuss the future research directions that we envision to explore.

Efficiency. Clearly, given its complexity, Ted requires further improvements on efficiency. Taking a closer look, we observe that as the Why-Not question becomes more general, the complexity increases, because the conditions that help reduce the number of cc-tuples to generate and process, are getting fewer. The worst case occurs when the Why-Not question asks *"Why-Not any tuple?"* in the case of an empty query result. Indeed, in this case, we have no choice but to first compute the cross product of all relations referenced in the query schema $\mathcal{S}_Q$ as the set of cc-tuples, to then identify the picky query operators for each such cc-tuple. To tackle the efficiency problem, we think of:

(1) Why-Not question restriction: We can start by proposing alternative Why-Not questions to the developer in order to narrow down the set of missing-answers. This translates into adding more constraints in the Why-Not question. Which questions to present (and in which order) depends on the one hand on statistics on data distribution in the source database that allow to estimate the benefit of adding a particular constraint to the Why-Not question in terms of estimated complexity. On the other hand, the suggested Why-Not question should not be too different from the original one. In a sense, we suggest modifications to the Why-Not question, similar in spirit to the adaptation of the Why-Not question of reverse skyline of queries in [27]. However, our optimization focus significantly differs as it is solely motivated by efficiency.

(2) Sampling-based Why-Not answer approximation: Data statistics as those mentioned above may also be used to determine a representative sample of the database to reduce the cc-tuple computations. Ideally, the representative sample still allows to detect all operator combinations OC_i of Ted's Why-Not answer, but with a number of occurrences reduced by a factor f. More specifically,

if the exact Why-Not answer were $N_1 * \Pi_{op_i \in OC_1} op_i + N_2 * \Pi_{op_i \in OC_2} + \ldots$, we aim for a sampling that will generate the approximate $N_1/f * \Pi_{op_i \in OC_1} op_i + N_2/f * \Pi_{op_i \in OC_2} + \ldots$.

(3) Execution environment: Ted can benefit from parallel computation capabilities of distributed platforms and MapReduce techniques to run concurrently the independent computation parts, e.g., processing partitions during cc-tuple computation or processing individual cc-tuples when identifying picky operators. Furthermore, given the types of queries that Ted generates, a column-store database system may be beneficial for runtime [28].

Query refinement. To avoid generating an overwhelmingly large number of alternative query rewritings, we plan to prune and rank solutions based on a cost model for possible query modifications that reflects the similarity to the original query. We will then generate query fixes that minimize this cost, which relates to work on edit distance. In this way, we favor rewritings most similar to the original query. Proposed algorithms will either return the top-k results w.r.t. the cost model or all rewritings up to a given cost threshold.

The Why-Not answers returned by our algorithms can serve as the starting point for the query rewriting process. Indeed, knowing the picky operators significantly narrows the search for a rewritten query, as we can first focus on finding "good" solutions that only affect this fragment of the query. Furthermore, TedExplain's Why-Not answer polynomial indicates what is the shortest picky operator combination and thus can be used to reduce the number of changes performed to achieve our goal.

6. CONCLUSIONS AND OUTLOOK

This thesis addresses the problem of explaining missing-answers for SQL queries and suggesting fixes to them to obtain the desired results. The current contributions consist in two algorithms computing query-based explanations, necessary formalizations, implementation and experimental evaluation. Currently, we are working on the efficiency aspects. Next, we will address the problem of proposing appropriate query rewritings. All methods will be implemented and evaluated, in order to properly validate our theoretical contribution and to provide relevant results in practice.

In the long term, our work could be extended to fit not only query but wider data transformations, including more complex manipulations. In this way, whole systems unexpected behaviour can be explained, which has also the benefit of reinforcing user trust, satisfaction, and acceptance of the system [30]. In a world where information and important decisions are derived and made upon processing Big Data, these properties seem to be more important than ever before.

7. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] B. Alexe, L. Chiticariu, and W.-C. Tan. Spider: a schema mapping debugger. In *PVLDB*, pages 1179–1182, 2006.

[3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *PVLDB*, 5(4):346–357, 2011.

[4] N. Bidoit, M. Herschel, and K. Tzompanaki. Immutably answering why-not questions for equivalent conjunctive queries. In *TaPP (to appear)*, 2014.

[5] N. Bidoit, M. Herschel, K. Tzompanaki, et al. Query-based why-not provenance with nedexplain. In *EDBT*, pages 145–156, 2014.

[6] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[7] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[8] A. Chapman and H. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[9] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[10] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[11] C. Curino, H. J. Moon, and C. Zaniolo. Graceful database schema evolution: the prism workbench. *PVLDB*, 1(1):761–772, 2008.

[12] C. V. Damásio, A. Analyti, and G. Antoniou. Provenance for sparql queries. In *ISWC*, pages 625–640. 2012.

[13] C. V. Damásio, A. Analyti, and G. Antoniou. Justifications for logic programming. In *Logic Programming and Nonmonotonic Reasoning*, pages 530–542. 2013.

[14] J. Danaparamita and W. Gatterbauer. QueryViz: helping users understand SQL queries and their patterns. In *EDBT*, pages 558–561, 2011.

[15] J. N. Foster, T. J. Green, and V. Tannen. Annotated xml: queries and provenance. In *SIGMOD-SIGACT-SIGART*, pages 271–280, 2008.

[16] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[17] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[18] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *ACM SIGMOD-SIGACT-SIGART*, pages 31–40, 2007.

[19] J. R. Groff, P. N. Weinberg, et al. *SQL: the complete reference*, volume 2. 2002.

[20] T. Grust, F. Kliebhan, J. Rittinger, and T. Schreiber. True language-level sql debugging. In *EDBT*, pages 562–565, 2011.

[21] Z. He and E. Lo. Answering why-not questions on top-k queries. In *ICDE*, pages 750–761, 2012.

[22] M. Herschel. Wondering why data are missing from query results?: ask conseil why-not. In *CIKM*, pages 2213–2218, 2013.

[23] M. Herschel and T. Grust. Transformation lifecycle management with nautilus. In *VLDB-QDB*, 2011.

[24] M. Herschel and M. A. Hernández. Explaining missing answers to SPJUA queries. *PVLDB*, 3(1-2):185–196, 2010.

[25] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[26] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *Journal of the ACM*, 31(4):761–791, 1984.

[27] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *ICDE*, pages 973–984, 2013.

[28] A. S. Kanade and A. Gopal. Choosing right database system: Row or column-store. In *ICICES*, pages 16–20, 2013.

[29] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. SnipSuggest: Context-aware autocompletion for SQL. *PVLDB*, 4(1):22–33, 2010.

[30] B. Y. Lim, A. K. Dey, and D. Avrahami. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *SIGCHI*, pages 2119–2128, 2009.

[31] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.

[32] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[33] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD*, pages 245–256, 2009.

[34] H. Park, R. Ikeda, and J. Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. *PVLDB*, 4(12):1351–1354, 2011.

[35] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD (to appear)*, 2014.

[36] S. Shah, S. Sudarshan, S. Kajbaje, S. Patidar, B. P. Gupta, and D. Vira. Generating test data for killing sql mutants: A constraint-based approach. In *ICDE*, pages 1175–1186, 2011.

[37] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD*, pages 15–26, 2010.

[38] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.

[39] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB*, pages 1006–1017, 2005.