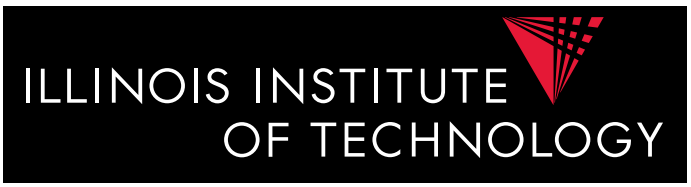


Automatic Generation and Ranking of Explanations for Mapping Errors

Seokki Lee, Zhen Wang, Boris Glavic, Renée J. Miller

IIT DB Group Technical Report IIT/CS-DB-2015-01

2015-02

http://www.cs.iit.edu/~dbgroup/

Automatic Generation and Ranking of Explanations for Mapping Errors

Seokki Lee
Illinois Institute of Technology
slee195@hawk.iit.edu

Zhen Wang
Amazon

Boris Glavic
Illinois Institute of Technology
bglavic@iit.edu

Renée J. Miller
University of Toronto
miller@cs.toronto.edu

ABSTRACT

Data transformation is facilitated by the use of visual and logical specifications of mappings between schemas. While easy to use, mappings are hard to design. Many techniques have been proposed to help users understand and refine mappings. However, once an error in transformed data has been identified, these systems at best provide low-level data-flow style tracing or query language facilities to help a mapping developer trace through the massive space of possible reasons for the error. In this work, we present an approach for systematically exploring the space of potential explanations (causes) for errors in transformed data. Our system leverages data provenance in combination with information about the mapping to automatically generate possible explanations. Since the number of potential explanations for a set of data errors is exponential in the size of the set it is unfeasible to present all possible explanations to a user - both from a usage and performance point of view. We address this problem by developing novel ranking mechanisms that allow us to present more likely explanations first. Even though the ranking problem is NP-hard in general, we demonstrate how to improve performance by spliting the problem into subproblems that can be solved independently and by pruning the search space using provable upper and lower bounds for the score of partial solutions. Our experimental evaluation confirms that, by applying these optimizations, we can generate explanations for large databases, complex mappings, and large sets of errors within reasonable time.

1. INTRODUCTION

Data transformation is a notoriously hard problem, one on which we have made great strides with the adoption of schema mappings, declaractive constraints specifying the relationship between data in a source and target schema. Schema mappings are the core abstraction driving many solutions for problems such as data integration, data exchange, data coordination, and mapping composition (used in schema evolution and other applications). Many systems have been proposed to create mappings between heterogeneous schemas (Clio [31], HePTox[8], ++Spicy [27, 26], BEA AquaLogic [7], to name a few). And once mappings have been designed or discovered, their complexity (and the subleties of complex heterogeneous data) has led to a host of techniques to help mapping developers in understanding or refining mappings (e.g., [1, 4]). These techniques use examples and other methods to illustrate the effect of choices such as grouping or nesting on the resulting mapping. Other approaches provide low-level debugging or tracing of mapping executions [2, 13]. In Spider, routes, are used to visualize and explain how source data flows to a target, together with programming-language-style debugging facilities like breakpoints. Other approaches use data (or mapping) provenance to help mapping developers understand mappings [21, 34]. These techniques let power-users (mapping developers) form SQL (or SQL-like) queries over a mapping scenario to understand how a mapping is transforming data. If an error is identified in target data (that is, a target attribute containing an incorrect value), these systems provide a programming infrastructure to help a developer trace what source data or what mappings are involved in (or influence) the creation of the error. However, they do not suggest specific *causes* from either the source data or the mapping scenario that produce the error.

In *Vagabond*, we address this problem by developing an approach that automatically generates potential causes of errors for a user-specified set of attribute values in a transformed (target) instance. Errors may be caused by factors such as the misinterpretation of attribute semantics, misinterpretation of the semantics of a referential relationship between tables, or source instance errors. Our work is motivated by the observation that data owners or curators can recognize errors in transformed data, but they often are not experienced mapping developers who understand the semantics of mapping languages enough to know what has caused the error. We also increase the productivity of developers by helping them quickly find the most likely error causes.

EXAMPLE 1. *Consider the example scenario in Figure 1. The source schema stores information about employees, firms, customers, and addresses (where a single relation* `Address` *stores addresses of employees, companies and customers).*

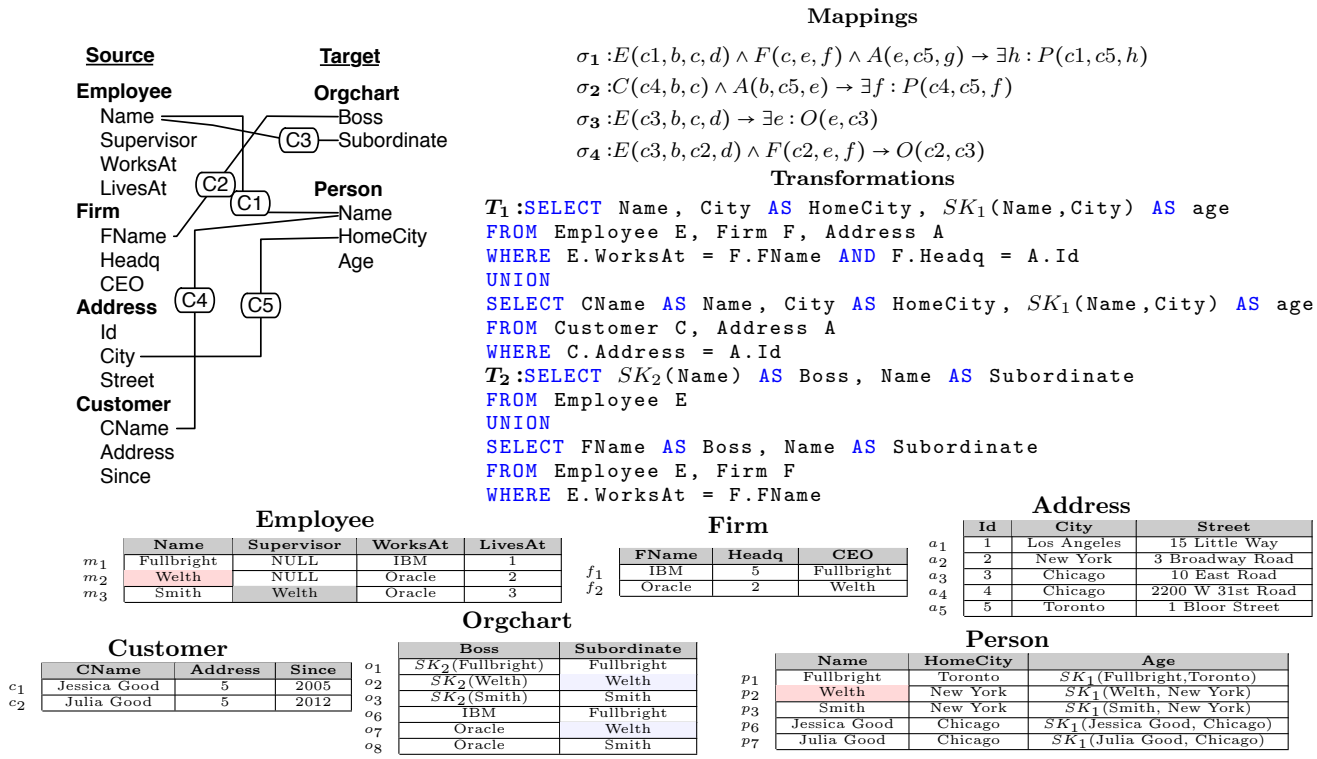


Figure 1: Example Data Exchange Scenario

This information is mapped to an `Orgchart` *relation and to a* `Person` *relation. Both employees and customers are mapped to persons. The mapping contains four source-to-target (ST) tuple-generating-dependencies (TGDs). The two SQL transformations shown in this figure represent a possible implementation of the exchange between the source and target schema. Assume that a data scientist realizes that the value "Welth" appearing in the* `Person` *relation (marked in red) is wrong. Call this error e_0. Note that this type of error does not manifest in any violations of the mapping or schema constraints. The incorrect value could have been* **caused** *by different components of the mapping scenario. For instance, this "Welth" target attribute value could have been copied from erroneous source value(s) - in the example, this is the single value $m_2.Name$. Note that this value has been copied to the target based on correspondence C_1. Thus, another potential cause is that this correspondence is incorrect. Indeed, these are just two of several possible causes for this error in the target data.*

1.0.1 Causes for Mapping Errors

Mapping errors often only become apparent once an erroneous target instance has been generated - after correspondences, mappings, and transformations have already been designed. For instance, a data scientist may notice unexpected missing (labelled null) values in a target attribute. Understanding what caused an incorrect target attribute value (an *error*) is a time-consuming and complex task. Three main factors contribute to the complexity: (1) different causes for an error may have the same or similar effects; (2) there are many possible sources of errors: the data, correspondences, mappings, and transformations; (3) a user (especially one who is not an experienced mapping developer) may not be aware of all potential types of errors that could have caused an observed error. Our first contribution is a classification of *types of causes* for target instance errors and a formalization of their effects on a target instance (called *coverage*). We call λ a *cause* for a target instance error e if the error $e \in coverage(\lambda)$. The coverage may include target instance values that the user deems to be correct. We refer to this subset of the coverage as the *side effects*. Note that we do not assume that the error set provided by a user is complete. But the explicit computation of the side effects allows the user to decide if some side effects are spurious (i.e., they are actually also errors).

Example 2. *For error e_0, one potential cause (λ_1) is that the source attribute value $m_2.Name$ is incorrect. This is an example of a specific type of cause called a Copy Data Error. The coverage of λ_1 is $\{p_2.Name, o_2.Subordinate, o_7.Subordinate\}$, hence $\{o_2.Subordinate, o_7.Subordinate\}$ are the side effects. A user can examine this set and decide if any of these side effects are themselves errors. A second potential cause (λ_2) is that the correspondence C_1 is wrong (a Correspondence Error). The side effects of λ_2 are $\{p_1.Name, p_3.Name\}$.*

1.0.2 Automatic Generation of Causes

A formalization of error causes can aid a user in debugging target instance errors. However, trying to manually enumerate all potential causes and their coverage is already cumbersome and error-prone for a single erroneous target value let alone for a large set of errors in a large target instance generated by a complex scenario. In this case, it is virtually impossible to manually generate all potential causes. Our second contribution addresses this problem. Based on

a classification of cause types, we study how to automatically generate all potential causes for a given set of errors, determine their coverage and side effects, and present them to the user.

EXAMPLE 3. *For e_0, we would generate seven causes. Intuitively, $\lambda_1 + \lambda_2$ from Ex. 2; λ_3 indicating that the mapping σ_1 is superfluous and should be removed: λ_4 indicating that an incorrect value in source attribute $f_2.FName$ caused m_2 to (incorrectly) join and appear in the output of mapping σ_1; λ_5, indicating that an incorrect value in source attribute $a_2.Id$ caused m_2 to appear in the output of mapping σ_1; and $\lambda_6 + \lambda_7$ indicating that the join between Firm and Employee (variable c in the mapping) or Firm and Address (variable e) is wrong, i.e., the mapping should not join these relations at all or should use a different join path. Each of these causes will have different side effects (e.g., λ_3 may have a large number of side effects as the entire mapping is removed, while for this example, the side effects of λ_1 is smaller).*

1.0.3 Ranking of Causes

Vagabond aides a user in debugging by automatically generating potential causes for target instance errors identified by the user. However, we face the problem that the number of causes for a set of errors is exponential in the number of errors. The user would have to explore a large search space of causes. Furthermore, we may not be able to enumerate all potential causes in reasonable time. We address this problem by developing incremental ranking techniques for causes. The rationale behind ranking is that we want to be able to present more likely causes first and, thus, allow the user to find the correct cause for a set of errors without having to browse through the whole exponentially large solution space. However, ranking is only useful if the scoring function that determines the rank of a solution is chosen wisely. In this work, we consider two scoring functions: ranking on side-effect size (causes that invalidate large parts of the target instance that the user deems to be correct are less likely to be correct) and number of causes needed to cover the error set (Occam's razor). Surprisingly, we show that producing the top-k ranked causes may be more efficient than just outputting all potential causes in some order, because we can avoid generating a cause if it can be determined upfront that it will not be ranked high. However, to be able to apply this pruning technique we need *incremental* ranking algorithms that do not require us to pre-generate the complete solution space.

EXAMPLE 4. *Continuing with Example 3, assume we want to rank causes λ_1 and λ_3 based on their side-effect size. We showed in Ex. 2, that cause λ_1 has a side-effect size is 2 and that cause λ_3 has a side-effect size of 8. Hence, Cause λ_1 would be ranked higher.*

As our third contribution we study the complexity of ranking (NP-hard), develop an incremental ranking algorithm based on A^*-search that exploits a lower bound property (which we prove for the score of partial solutions), and introduce a novel partitioning scheme that allows us to divide the problem into sub-problems of smaller size that can be ranked independently. While ranking for a sub-problem is still NP-hard, combining the sub-problem rankings into a global ranking is efficient ($O(n \cdot log(n))$).

The user interface of Vagabond has been demonstrated in [22]. In this paper, we formalize the cause generation and ranking problems, presenting efficient solutions for both. The main contributions of this work are the following.

- We present a classification of causes for target instance errors and study their coverage and side effects.
- We present an approach to automatically and systematically explore the space of possible causes for a given set of errors. Our approach is not tied to a particular mapping system and type of solution (e.g., universal or core solutions). It can operate over the output of any data exchange system that uses st-tgds (the most widely used mapping formalism) as long as SQL queries implementing the mappings are annotated with the mappings they implement.
- We develop efficient ranking techniques for causes and investigate several optimizations of the ranking schemes that make the approach, being NP-hard in general, applicable to real-world problems. Specifically, we prove and exploit a lower bound for scores of partial solutions. We show how to partition the problem into smaller subproblems which can be solved independently and how to combine per-partition solutions efficiently into a global solution.
- We present an efficient implementation of the algorithms in the *Vagabond* [22] middleware system which runs over the *TRAMP* [21] provenance management system. Vagabond leverages standard data provenance definitions and could equally have been developed over other provenance management systems such as Orchestra [24].
- Our experiments, using real scenarios and the *iBench* [5] mapping benchmark to produce a large variety of mapping scenarios, confirm the efficiency and effectiveness of the proposed solutions.[1]

The remainder of this paper is structured as follows. We discuss related work next, then introduce necessary background in Section 3. Section 4 introduces our cause model and generation algorithms. Section 5 presents new techniques for efficiently generating top ranked causes. Our implementation and experimental results are presented in Sections 6 and 7. We conclude in Section 8.

2. RELATED WORK

Three lines of work are related to our approach: approaches for debugging schema mappings, ranking and top-k query processing, and causality as well as generating explanations for missing answers and other integration tasks such as data fusion [15] or cleaning [10]. Note that we generate explanations (causes) for errors in the target instance of a consistent data exchange scenario. Thus, our work is orthogonal to approaches for fixing inconsistencies such as constraint violations in the data exchange scenario.

2.1 Debugging and Understanding Mappings

The need for simplifying the mapping creation process and to support the user in debugging mapping scenarios has been addressed by two major lines of work. *Muse* [1] and *Eirene* [4, 3] enable a mapping developer to create or refine schema mappings by choosing between example instances generated by the system which illustrate the effect of different choices in the creation or refinement process. These approaches are orthogonal to our work. The second line of work uses provenance information to aid a developer

[1] All proofs are presented in Appendix A

debugging mappings. *Spider* [13] lets a developer trace mapping executions (called routes) that lead to the generation of certain tuples in the target. A route combines data and mapping provenance. *MXQL* [34] generates provenance for a data exchange scenario eagerly during the execution of a transformation. The generated target instance is enriched with mapping provenance and provenance that relates source to target schema elements. The *TRAMP* [21] system generates several types of provenance (data, transformation, and mapping) on-the-fly when the developer requests this information through one of the SQL extensions of the system. The data exchange scenario is stored in the DB to allow for queries that explore this data.

All systems mentioned above assume that the mapping developer is aware of what could cause a target data error. Automatic generation of error causes makes our system much more accessible for non-power users and can improve the productivity of power-users, because the user only has to specify which parts of the target instance she considers to be erroneous and then can start directly to explore the space of potential causes without the need to generate the potential causes manually.

2.2 Ranking

Ranking sets of causes has some similarity with top-k query processing and ranking query results. However, as we will demonstrate later, the main difference is that while in many top-k query processing approaches materializing the query result is an option, this is not true for ranking error causes because the space of potential causes is too large. Furthermore, ranking algorithms such as Fagin's algorithm [16], rely on monotonicity properties of scoring functions to prune parts of the search space. Unfortunately, these properties do to not hold for the scoring functions used in this paper. However, the general idea of avoiding traversal of the complete search space by using properties of the scoring function are also valid for ranking causes. Similar to the J^* algorithm for top-k join queries [32], our solution is also based on the general idea behind the A^* search algorithm, but in contrast to J^* we are dealing with non-montone scoring functions.

2.3 Causality and Explanations

Our work shares motivation with approaches for finding causes for answers to a query [11, 28, 33] and finding explanations (e.g., for data fusion [15] or cleaning [10] decisions). Similar to these approaches we are searching for evidence in the input of some transformation that justifies a certain output of interest. However, most of these approaches consider only data to be a potential cause while we also consider other scenario elements such as correspondences of mappings. There exists an additional fundamental differences between the problem we are addressing and work on causality and explanations. In our work, we are interested in the side effects that causes have on the target instance, i.e., if we assume a cause to be correct then this will imply that other target instance elements will be incorrect. This has not been explored in the work on causality. This aspect of our problem is more similar to computing side effects of view updates, i.e., if the user considers a target attribute value to be incorrect then this essentially means that we should change the value of this attribute or remove the tuple it resides in. However, in contrast to the view update problem we do not know what the resulting value should be. In addition to top-k query processing, the idea of ranking solutions based on, e.g., their side effects has also been explored for finding missing answers [23] and answering queries over hypothetical scenarios [29]. These approaches rely on constraint solvers to identify an optimal solution. In contrast, our approach for ranking solutions is more closely related to top-k query processing and informed search. Note that while these approaches return one optimal solution, our algorithm allows an incremental traversal of the search space in decreasing optimality. This is a critical feature for our application domain where a user can iteratively explore different solutions until the correct one is found.

3. PRELIMINARIES AND BACKGROUND

3.1 Data Exchange

3.1.1 Schema Mappings

A *schema* $\mathbf{R}$ is a set of relation schemas $\mathbf{R}_1, \ldots, \mathbf{R}_n$. An *instance* R for a schema $\mathbf{R}$ is a set of relations $R_1, \ldots, R_n$ with the same arity as the relation schemas in $\mathbf{R}$. For simplicity, we assume that all attribute values are drawn from a single infinite domain $\mathbb{D}$. A schema mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ consists of two schemas $\mathbf{S}$ and $\mathbf{T}$ (called *source* and *target* schema) and a mapping Σ. We limit the discussion to *source-to-target tuple generating dependencies* (*st-tgds*). An st-tgds is a formula $\forall \bar{x} : \phi(\bar{x}) \rightarrow \exists \bar{y} : \psi(\bar{x}, \bar{y})$ where $\phi(\bar{x})$, and $\psi(\bar{x}, \bar{y})$, are conjunctions of atomic formulas over the source, respective target, schema. Atomic formulas are expressions over the relations in a schema. We often omit universal quantifiers in mappings. For a mapping σ we use $var(R.A)$ to denote all the variables used for attribute A in an atom for relation R. Likewise $attr(v)$ for an variable v denotes all the attributes where the variable v appears in a mapping. For example, given source relation $R(A_1, A_2)$ and the mapping $R(a, b) \wedge R(b, a) \rightarrow \exists c : T(a, c)$, $attr(b)$ is $\{R.A_1, R.A_1\}$ and $var(R.A_1) = \{a, b\}$.

3.1.2 Solutions

We use I (respectively J) to refer to an instance of the schema $\mathbf{S}$ (respectively $\mathbf{T}$). An instance J is called a *solution* for a schema mapping $\mathcal{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$ with respect to an instance I, if $(I, J) \models \Sigma$, i.e., the instances fulfill all constraints in Σ. The values of J are drawn from the domain $\mathbb{D}$ and a sufficiently large set $\mathbb{N}$ of distinguished null values (called *labelled nulls*). A valuation θ for a mapping σ is an assignment from variables to constants so that $\sigma[\theta]$ (called grounded mapping) is fulfilled in (I, J). We use $(I, J) \models \sigma[\theta]$ to denote that grounded mapping $\sigma[\theta]$ holds in (I, J). Furthermore, we use $R(t) \triangleleft \sigma[\theta]$ to denote that $R(t)$ is a grounded atom in $\sigma[\theta]$ and $R(t) \triangleleft \psi[\theta]$ respective $R(t) \triangleleft \phi[\theta]$ to denote that the atom $R(t)$ is part of the grounded left-hand respective right-hand side of a mapping.

3.1.3 Data Exchange Scenarios

In addition to schemas, mappings and instances, our cause generation techniques use additional information about a data exchange setting including *attribute correspondences* and *transformations*. Formally, an attribute correspondence is a tuple $C = (\bar{x}, \bar{y})$ where $\bar{x}$ a list of source attributes and $\bar{y}$ is a list of target attributes of the same arity. Data exchange systems which work with st-tgds (or similar

declarative specifications of a schema mapping) derive executable *transformations* (in, e.g., Java, SQL, or XQuery) from the st-tgds to be able to create a target instance (a solution) from a given source instance. Most systems aim to produce solutions with good properties such as universal solutions [18] or cores [17]. Here, we focus on SQL transformations. Our techniques are agnostic to the choice of solution that the transformation creates. We consider correspondences and transformations as part of a data exchange scenario, because (1) they are additional sources of errors and (2) to support SQL transformations generated by different data exchange systems. We require an explicit specification (MAP) that records interdependencies between the elements of a data exchange scenario. We use $X(l)$ to denote scenario elements of type X that are related to element l according to MAP, e.g., $\Gamma(\sigma)$ denotes the correspondences used by a mapping σ. The entries of MAP can either be inferred from the scenario elements or produced with minor changes by an exchange system.

DEFINITION 1 (DES). *A data exchange scenario (DES) is a tuple* $\mathbb{M} = (\mathbf{S}, \mathbf{T}, \Sigma, \Gamma, I, J, \mathcal{T}, \text{MAP})$ *where* Σ *is a set of st-tgds,* $(I, J) \models \Sigma$, Γ *is a set of correspondences,* $\mathcal{T}$ *are the transformations implementing the mapping, and* MAP *represents the relationships between scenario elements.*

EXAMPLE 5. *Consider the data exchange scenario shown in Figure 1. The correspondences (shown as labelled lines* C_1 *to* C_5 *in the schemas) relate employee and customer names to person names, address cities to person homeCities, a firm* `FName` *to the* `Boss` *attribute, and finally employee names to the* `Subordinate` *attribute. These correspondences are used by the four mappings of the scenario. The SQL transformation* T_1 *implements* σ_1 *and* σ_2*, while* T_2 *implements* σ_3 *and* σ_4*. Of course other transformations are possible.*

3.2 Provenance

We now introduce several types of provenance that are used to define the cause of an error. The types of provenance we use are examples of well-known provenance types from the literature (and are expressible using semiring annotations with functions to express mappings [24]). We refer the reader to Glavic et al. [19], Cheney et al. [12], and Karvounarakis et al. [24] for an overview on database provenance theory and computation.

3.2.1 Copy Provenance

Copy provenance models from which attribute values in the source a value in the target has been copied. We use a triple (R, t, A) (called a *cell*) to denote attribute A in tuple t from relation R and $Cells(I)$ to denote all such triples for an instance I. The copy provenance $\mathcal{C}(e)$ of a cell (R, t, A) in the target instance is the set of cells from the source from which values have been copied to this cell. Here we give a definition based on the mappings of a data exchange scenario $\mathbb{M}$. Given a valuation θ for a mapping σ and a grounded atom $R(t)$ in $\sigma[\theta]$ we use $var((R, t, A), \theta)$ to denote the variable(s) used by the ungrounded version of σ at attribute A in the atom corresponding to $R(t)$. The copy provenance of a target cell x consists of all attributes in a grounded mapping that use the same variable that was replaced by the value of x. For instance, for a mapping $\forall a, b : R(a, b) \wedge S(b) \rightarrow T(b)$, the copy provenance of a cell in the single attribute of T will contain cells from the first attribute of R and the single attribute of S.

$$\mathcal{C}(R.t.A) = \{R'.t'.A' \mid \exists \sigma \in \Sigma, \theta : (I, J) \models \sigma[\theta] \wedge R(t) \triangleleft \phi[\theta] \wedge R'(t') \triangleleft \psi[\theta] \wedge var(R.t.A, \theta) \cap var(R'.t'.A', \theta) \neq \varnothing\}$$

Similarly, we define the restriction of the copy provenance according to one grounded tgd $\sigma[\theta]$:

$$\mathcal{C}(R.t.A, \sigma[\theta]) = \{R'.t'.A') \mid R(t) \triangleleft \phi[\theta] \wedge R'(t') \triangleleft \psi[\theta] \wedge var(R.t.A, \theta) \cap var(R'.t'.A'), \theta) \neq \varnothing\}$$

EXAMPLE 6. *Returning to the erroneous value "Welth" in cell* $e_0 = (Person.p_2.Name)$ *from Figure 1. This value has been copied from cell* $e_s = (Employee.m_2.Name)$ *in the source instance. The only valid valuation is of mapping* σ_1*, specifically:* $E(m_2) \wedge F(f_2) \wedge A(a_2) \rightarrow P(p_2)$*. The variable* c_1 *bound to* e_0 *by this valuation is also bound to* e_s *(and to no other cell). Thus,* $\mathcal{C}(e) = \{e_s\}$.

3.2.2 Influence Provenance

Influence provenance models the tuples in the source that were used to derive a tuple in the target. The influence provenance of a target tuple t contains all source tuples that were used to derive this tuple. For mappings, this corresponds to all tuples used in valuations that have t on their righthand side.

$$\mathcal{PI}(t) = \{t' \mid \exists \sigma \in \Sigma, R' \in \mathbf{S}, \theta : (I, J) \models \sigma[\theta] \wedge R(t) \triangleleft \psi[\theta] \wedge R'(t') \triangleleft \phi[\theta]\}$$

EXAMPLE 7. *The influence provenance of the tuple* p_2 *from the running example in Figure 1 is the set containing the tuples* e_2*,* f_2*, and* a_2*, because the only grounded mapping containing* p_2 *is* $\sigma_2[\theta] = E(e_2) \wedge F(f_2) \wedge A(a_2) \rightarrow P(p_2)$.

3.2.3 Mapping Provenance

The mapping provenance $\mathcal{M}(t)$ of a tuple t from a target instance J contains all mappings from Σ that created the tuple t. The mapping provenance is used to determine which mappings are potential causes for an erroneous target cell. To be precise, let t be a tuple in a relation R in J, then a tgd $\sigma = \phi(\bar{x}) \rightarrow \psi(\bar{x}, \bar{y})$ belongs to the mapping provenance of t if there exists a valuation θ so that $\sigma[\theta]$ is fulfilled by (I, J) and the atom $R(t)$ is in the grounded right-hand side: $R(t) \triangleleft \phi[\theta]$.

$$\mathcal{M}(t) = \{\sigma \mid \sigma \in \Sigma \wedge \exists \theta : (I, J) \models \sigma[\theta] \wedge R(t) \triangleleft \psi[\theta]\}$$

EXAMPLE 8. *Tuple* o_6 *from Figure 1 was produced by mapping* σ_4 *(and only this mapping). Thus, the mapping provenance of* o_6 *is* $\{\sigma_4\}$.

4. ERRORS AND CAUSES

.We now formalize errors, present the types of error causes that we consider in this work, and state the cause generation problem, i.e., given a set of errors find all possible combinations of causes that fully explain these errors. Each cause explains an error by assuming that one or more elements of the DES are erroneous in a certain way.

4.1 Target Instance Errors

The input to our framework is a (not necessarily complete) set of target cells that are considered to be incorrect.

DEFINITION 2 (ERRORS). *An error e for a DES $\mathbb{M}$ is an element of $Cells(J)$. E is a set of errors.*

The semantics of an error is that the value at this cell of the target instance is incorrect. For example, error $e_0 = Person.p_2.Name$ in Figure 1 would indicate that the value "Welth" is incorrect for this cell. Note that such an error does not state that the value "Welth" would be incorrect if used in other target cells. A target cell e may have been derived by grounding more than one tgd or through multiple groundings of the same tgd. A cause for e should explain what went wrong in each of these derivations.

DEFINITION 3 (ERROR GROUNDINGS). *Let $e = R.t.A$ be an error for a DES $\mathbb{M}$. We define the groundings $\Theta(e)$ for e as the set of grounded tgds in (I, J) that have a grounded atom corresponding to $R.t$:*

$$\Theta(e) = \{\sigma[\theta] \mid \sigma \in \Sigma \wedge (I, J) \vDash \sigma[\theta] \wedge R(t) \triangleleft \sigma[\theta]\}$$

For an error set E, we use $\Theta(E)$ to denote the union of the sets of groundings for all errors in the set:

$$\Theta(E) = \bigcup_{e \in E} \Theta(e)$$

Note that the set of error groundings can be determined by combining the mapping provenance of a tuple t with the influence provenance of t. The influence provenance gives us a valuation θ and the mapping provenance allows us to construct the grounded tgds corresponding to this valuation.

4.2 Causes

A *cause* is a possible reason for one grounding of an error. As mentioned before, a cause being correct may imply other parts of the target instance to be incorrect. We call the part of the target instance invalidated by a cause λ the coverage of λ and use O to denote the data exchange scenario elements that the cause deems to have been the reason the error. The side effect of a (set of) cause(s) is the set of target cells in the coverage that are not considered as erroneous by the user (i.e., not in error set E).

DEFINITION 4 (CAUSE). *Given a grounding $\sigma[\theta]$ for an error $e = R.t.A$ and data exchange scenario $\mathbb{M}$, a cause λ for $\sigma[\theta]$ according to e is a tuple $\lambda = (T, O)$ where T is the type of cause (one of a fixed set of types $\mathbb{T}$ defined below) and O is a set of elements from the data exchange scenario $\mathbb{M}$. We write $\lambda \rightsquigarrow (\sigma[\theta], e)$ to denote that λ is a cause for a grounded tgd $\sigma[\theta] \in \Theta(e)$ according to e.*

For a cause λ, the coverage $\text{COVER}(\lambda)$ is the subset of the cells from the target instance J for which there exists a grounding that is covered by the cause. The side effects $\text{SE}(\lambda)$ of a cause are the cells it covers in addition to e.

DEFINITION 5 (COVERAGE AND SIDE EFFECTS). *Let λ be a cause for e. The coverage and side effects of λ are:*

$$\text{COVER}(\lambda) = \{e' \mid \exists \sigma[\theta] \in \Theta(e) : \lambda \rightsquigarrow (\sigma[\theta], e')\}$$
$$\text{SE}(\lambda, e) = \text{COVER}(\lambda) - \{e\}$$

We now define the types of causes considered by our approach. For a given data exchange scenario $\mathbb{M}$, error e, and grounded tgd $\sigma[\theta] \in \Theta(e)$, we define $\mathcal{T}(\mathbb{M}, e, \sigma[\theta])$, the set of all possible causes of type $\mathcal{T}$ for $\sigma[\theta]$ according to e in $\mathbb{M}$. Our approach is extensible to additional types.

4.2.1 Copy Data Error ($\mathcal{T}_{SC}$)

A copy data error $\lambda = (\mathcal{T}_{SC}, O_I)$ explains an error e for a grounding $\sigma[\theta]$ by assuming that the set of cells O in the source from where the value at e has been copied are incorrect. Hence, $O = \mathcal{C}(e, \sigma[\theta])$ is precisely the copy provenance of e for this grounding as defined in Section 3.0.3. If cell e' in the source is incorrect then that implies that all cells in the target where the value from e' has been copied are also incorrect. Thus, the coverage of λ contains all target cells which have been copied from a cell in O, i.e., cells that have at least one cell from O in their copy provenance. There is exactly one copy data error cause for each error grounding $\sigma[\theta]$ unless the value at e has been generated by the tgd ($\mathcal{C}(e, \sigma[\theta]) = \varnothing$), meaning the tgd uses a constant value to populate the target attribute.

$$\mathcal{T}_{SC}(e, \sigma[\theta], \mathbb{M}) = \begin{cases} \{(\mathcal{T}_{SC}, O_I)\} & \text{if } \mathcal{C}(e, \sigma[\theta]) \neq \varnothing \\ \varnothing & \text{else} \end{cases}$$
$$O_I = \{e' \mid e' \in \mathcal{C}(e, \sigma[\theta])\}$$
$$\text{COVER}(\lambda) = \{e' \mid \exists e'' \in O : e'' \in \mathcal{C}(e')\}$$

4.2.2 Correspondence Error ($\mathcal{T}_C$)

An erroneous value $e = R.t.A$ may have been copied by a grounded tgd $\sigma[\theta]$ from a source cell, because the tgd σ is based on correspondence(s). So $O = \{\gamma = (R'.A', R.A) \mid \sigma \in \Sigma(\gamma)\}$. Often O contains a single correspondence unless several values are mapped to A (for example, $S(x, x) \rightarrow R(x)$). [2] Such a cause implies that all tgds using one of these correspondences are erroneous and, in turn, also the transformations implementing these mappings are incorrect. Thus, a *correspondence error* invalidates correspondences, mappings, and transformations. The set O_Γ of correspondences can be determined of follows. A correspondence γ maps to e according to $\sigma[\theta]$ if σ is in the mapping provenance of t and σ uses γ. Based on O_Γ, we can determine the affected mappings (O_Σ) and transformations ($O_\mathcal{T}$) from the DES. The coverage of λ contains all cells $R.t'.A$ for tuples t' that were produced by a tgd σ' using a correspondence from O, i.e., that have this σ in their mapping provenance $\mathcal{M}(t')$.

$$\mathcal{T}_C(e, \sigma[\theta], \mathbb{M}) = \begin{cases} \{\lambda = (\mathcal{T}_C, O_\Gamma \cup O_\Sigma \cup O_\mathcal{T})\} & \text{if } O_\Gamma \neq \varnothing \\ \varnothing & \text{else} \end{cases}$$
$$O_\Gamma = \{\gamma = (R'.A', R.A) \mid \sigma \in \Sigma(\gamma)\}$$
$$O_\Sigma = \{\sigma' \mid \sigma' \in \mathcal{M}(t) \wedge \sigma' \in \Sigma(O_\Gamma)\}$$
$$O_\mathcal{T} = \{T \mid T \in \mathcal{T}(O_\Sigma)\}$$

$$\text{COVER}(\lambda) = \{R.t'.A) \mid \mathcal{M}(t') \cap O_\Sigma \neq \varnothing \wedge \exists C \in O_\Gamma : C = (R'.A', R.A)\}$$

4.2.3 Superfluous Mapping Error ($\mathcal{T}_{SM}$)

A superfluous mapping error explains each grounding $\sigma[\theta]$ by assuming that the tgd σ should be removed. Hence, $O = \{\sigma\}$ and the coverage is all cells of tuples generated by

[2] Not all target cells are necessarily produced by copying value(s) from the source. For example, this is not the case when a constant value is generated by the tgd or the value is a labeled null.

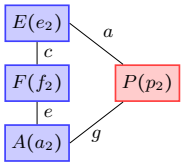


Figure 2: Join Graph for a Grounding of σ_1

σ.

$$\mathcal{T}_{SM}(e, \sigma[\theta], \mathbb{M}) = \{\lambda = (\mathcal{T}_{SM}, O_\Sigma \cup O_\mathcal{T})\}$$
$$O_\Sigma = \{\sigma\}$$
$$O_\mathcal{T} = \{T \mid T \in \mathcal{T}(O_\Sigma)\}$$
$$\textsc{Cover}(\lambda) = \{R.t'.A' \mid \sigma \in \mathcal{M}(t')\}$$

4.2.4 Join Data Error ($\mathcal{T}_{JD}$)

A join data error $\lambda = (\mathcal{T}_{JD}, O)$ for an error e occurs if an incorrect value for a join attribute caused incorrect tuples to be joined. The incorrect join caused the value at e to be copied from the "wrong" tuple. For instance, consider a tgd $R(a, b), S(b, c) \rightarrow T(a, c)$ over relations $R(A_1, A_2)$ and $S(A_3, A_4)$. An incorrect value in attribute A_2 might cause the A_4 attribute value to be copied to the target from a wrong tuple. We define a graph structure for grounded tgds that allows use to determine such cells for a given grounding.

The join graph $G(\sigma[\theta])$ of a grounded tgd $\sigma[\theta]$ contains a node for each grounded atom in $\sigma[\theta]$ and an edge labeled V between grounded atomes $R(t)$ and $R'(t')$ if the set of variables V was used both in the atoms corresponding to R and R'. The edges in a join graph represent join conditions (if the edge is between source atoms) and copying of values to the target (if the edge is between a source and a target atom). For instance, consider the join graph for the grounded tgd $\sigma_2[\theta] = E(e_2) \wedge F(f_2) \wedge A(a_2) \rightarrow P(p_2)$ from the running example (Figure 2). The edge between $E(e_2)$ and $F(f_2)$ indicates that these two atoms have been joined through variable c which was substituted with "Oracle".

Let $e = R.t.A$ be an error and $\sigma[\theta]$ a grounded tgd in $\Theta(e)$. Let $\textsc{SAtoms}(e, \sigma[\theta])$ denote the set of source atoms directly connected though attribute A with the atom $R(t)$, i.e., the source atoms from where the value at e has been copied. We use $\textsc{Path}(e, \sigma[\theta])$ to denote all paths in the join graph for $\sigma[\theta]$ that start in an atom in $\textsc{SAtoms}(e, \sigma[\theta])$, contain only source atoms not in $\textsc{SAtoms}(e, \sigma[\theta])$, and do not repeat atoms. Each edge in such a path represents a join between two tuples in the grounded tgd that may have caused an incorrect tuple to be joined and, thus, an incorrect value to be copied from an atom in $\textsc{SAtoms}(e, \sigma[\theta])$ to the target at the cell e. For each such path $p = R_1(t_1), R_2(t_2), \ldots, R_m(t_m)$, we create one join error for every edge in the path. Let $var(p, R'(t'))$ denote the set of variables on the label of the incoming edge to $R'(t')$ in the order of path p. We consider all cells in $attr(R'(t'), var(p, R'(t')))$ at positions of atom $R'(t')$ corresponding to variables in $var(p, R'(t'))$ as a potential join data error. If the values of one of these cells is incorrect, then incorrect tuples may have been joined leading to an incorrect value at e.

Example 9. *For example, for the join graph shown in Figure 2 and the error $e = Person.p_2.HomeCity$, the value at e has been copied from atom $A(a_2)$. There is one path starting at this atom $p = A(A_2), F(f_2), E(e_2)$ in the join graph. Thus, based on the labels e and c of the edges on the path, we would consider two source cells as join data errors: $E.e_2.WorksAt$ and $F.f_2.Headq$. Since in this example we have only one path with two edges, each labeled with a single variable, we would only generate these two join data errors.*

The coverage of a join data error $(\mathcal{T}_{JD}, O_I)$ contains all attribute values e' for which there exists a grounding $\sigma'[\theta']$ and an element $R'.t'.A'$ in O_I with the following property: the attribute A' corresponds to the variable on an incoming edge label for an atom $R'(t')$ on a path in $\textsc{Path}(e', \sigma'[\theta'])$. That is all errors e' for which there exists a grounding where $R'(t')$ is on a join path. In the definition of source join errors presented below we define a set $\textsc{PVars}(e, \sigma[\theta])$ that contains all pairs of atoms $R'(t')$ and variables v for which there exist a path p in $\textsc{Path}(e, \sigma[\theta])$ that contains an incoming edge to atom $R'(t')$ labelled with a set V' containing v.

$$\mathcal{T}_{JD}(e, \sigma[\theta], \mathbb{M}) = \{\lambda_x = (\mathcal{T}_{JD}, O_I(x)) \mid x \in \textsc{PVars}(e, \sigma[\theta])\}$$
$$\textsc{PVars}(e, \sigma[\theta]) = \{(R'(t'), v) \mid \exists p \in \textsc{Path}(R(t), \sigma[\theta]) : R'(t') \in p \wedge v \in var(p, R'(t'))\}$$
$$O_I(R'(t'), v) = \{R'.t'.A' \mid A' \in attr(R'(t'), v)\}$$
$$\textsc{Cover}(\lambda_x) = \{e' \mid \exists \sigma'[\theta'] \in \Theta(e') : \exists x' = (R'(t'), v') \in \textsc{PVars}(e', \sigma'[\theta']) : attr(x) = attr(x')\}$$

4.2.5 Join Mapping Error ($\mathcal{T}_{JM}$)

A join mapping error assumes that the error e for a grounding $\sigma[\theta]$ was caused by σ using an incorrect join path between source relations.

Example 10. *The tgd σ_1 from Figure 1 maps employee names and the addresses of the company they are working for to person tuples in the target. If the `HomeCity` attribute in the target is meant to store the city where an employee lives, then σ_1 uses an incorrect join path between the `Employee` and `Address` relations leading to incorrect `HomeCity` values in target tuples produced by σ_1. The correct way of joining these two relations might be to join on the `LivesAt` attribute.*

We again use the join graph for a tgd to determine from which atoms ($\textsc{SAtoms}(e, \sigma[\theta])$, introduced for join errors) in the left-hand side of a tgd the value at an error has been copied from. A join mapping error explains an error by assuming that one of these atoms has been joined incorrectly with another atom not in $\textsc{SAtoms}(e, \sigma[\theta])$ through a variable v and, thus, all values that have been copied from these atoms to the target by the tgd are erroneous. For a given error $e = R.t.A$ and grounded tgd $\sigma[\theta] \in \Theta(e)$, we use PVars introduced for join errors to find such atom-variable pairs. The coverage of a join mapping error $\lambda = (\mathcal{T}_{JM}, \{(R, v), \sigma, T\})$ are all cells e' that have been copied by tgd σ from any atom for which there exists a grounding $\sigma[\theta']$ such that $(R(t), v)$ is on a path in the join graph $\textsc{Path}(e', \sigma[\theta'])$. For example, consider a join mapping error $\lambda = (\mathcal{T}_{JM}, \{\sigma_1, (E, c)\})$ which explains $e = P.p_1.HomeCity$ in the running example by assuming that the join attribute `WorksAt` is not the right attribute to join on. The coverage of λ are the `HomeCity` attribute values from tuples p_1 to p_5.

EXAMPLE 11. *For instance, for $e = P.p_2.HomeCity$ the tgd σ_1 generated this value through a valuation θ and the sole atom in* $\text{SATOMS}(e, \sigma_1[\theta])$ *is* $A(a_2)$. *The join graph for* $\sigma_1[\theta]$ *is shown in Figure 2. The single path in* $\text{PATH}(e, \sigma[\theta])$ *is* $A(a_2), F(f_2), E(e_2)$. *Thus, the error could have been caused by the join between the* `Address` *and* `Firm` *relations using variable e or by the join between* `Employee` *and* `Firm` *using variable c.*

$$\mathcal{T}_{JM}(e, \sigma[\theta], \mathbb{M}) = \{\lambda_x = (\mathcal{T}_{JM}, O_\Sigma \cup O_{\text{AVARS}}(x)) \mid x \in \text{PVARS}(e, \sigma[\theta])\}$$

$$\text{PVARS}(e, \sigma[\theta]) = \{(R'(t'), v) \mid \exists p \in \text{PATH}(e, G(\sigma[\theta])) : R'(t') \in p \wedge v \in var(p, R'(t'))\}$$

$$O_{\text{AVARS}}(R'(t'), v) = \{(R', v)\}$$

$$O_\Sigma = \{\sigma\}$$

$$O_\mathcal{T} = \{T \mid T \in \mathcal{T}(O_\Sigma)\}$$

$$\text{COVER}(\lambda_x) = \{e' \mid \exists \sigma[\theta'] \in \Theta(e') : x \in \text{PVARS}(e', \sigma[\theta'])\}$$

The set of cause types presented cover a wide-range of real-world error causes, but is not complete. For instance, in a sense we assume minimality of causes. For example, we do not consider a combination of a join mapping error and a join error as a cause (for a single grounded tgd for an error in the target). We leave the exploration of additional types such as errors in the transformations for future work.

EXAMPLE 12 (CAUSE TYPES). *Consider the error $e = P.p_2.Name$ for the instance shown in Figure 1. Figure 3 shows all potential causes for this error. The tuple p_2 was generated by exactly one grounded tgd: $\sigma_1[\theta] : E(e_2) \wedge F(f_2) \wedge A(a_2) \rightarrow P(p_2)$ which makes the set of causes relatively small (seven). These are the seven causes explained intuitively in Example 3 in the introduction.*

4.3 Covering Cause Sets

For a given error set E and its groundings there are many possible sets of causes that explain all groundings for all errors in the set. We call such a cause set Λ, a *covering cause set* (*CCS*) for E.

DEFINITION 6 (CCS). *Given a set of causes Λ and a set of errors E, we define the coverage, side-effects, and full coverage as:*

$$\text{COVER}(\Lambda) = \bigcup_{\lambda \in \Lambda} \text{COVER}(\lambda)$$

$$\text{SE}(\Lambda, E) = (\bigcup_{\lambda \in \Lambda} \text{COVER}(\lambda)) - E$$

$$\text{FCOVER}(\Lambda) = \{e \mid \forall \sigma[\theta] \in \Theta(e) : \exists \lambda \in \Lambda : \lambda \rightsquigarrow (e, \sigma[\theta])\}$$

A set Λ of causes is called a covering cause set (CCS) for an error set E iff: $E \subseteq \text{FCOVER}(\Lambda)$

In Section 6, we discuss how to generate the set of all possible causes for each grounding $\sigma[\theta]$ for an error e from a set E. Each combination of one cause from each of these sets is a valid CCS for E. Note that these sets may overlap, so the total number of causes may be less than the sum of the number of causes for all error groundings. We call the problem of enumerating all valid CCS, the *Covering Cause Set Enumeration Problem*. A naive way to enumerate all CCS for an error set E is to generate all sets Λ_{ij} for each error e_i in E and grounding $\sigma[\theta]_{ij}$ in $\Theta(e_i)$, then compute the cross product, and create a set of causes from each tuple in the cross-product.

DEFINITION 7 (CCS ENUMERATION PROBLEM). *Let E be an error set. A solution to the covering cause set enumeration problem is the set $CCS(E)$ of all valid covering cause sets for E:*

$$CCS(E) = \{\Lambda \mid E \subseteq \text{FCOVER}(\Lambda)\}$$

Unfortunately, the number of potential CCS for a set of errors can be exponential in size. Thus, while generating all potential causes for a single error is efficient as long as access to provenance is efficient, we can not hope to find an efficient way of enumerating all potential CCS for a set of errors. We will address this problem in Section 5 by demonstrating how to produce the top-k CCS according to a scoring functions without enumerating all CCS.

PROPOSITION 4.1. *The size of $CCS(E)$ for an error set E can be exponential in the number of elements of E.*

5. RANKING OF CAUSES

Because of the exponential size of CCS enumerations, it is unfeasible to present a full CCS enumeration to the user. Typically, the CCS for an error set are not homogeneous, but differ in size, side effects, and other properties. We want to use these properties to rank CCS according to a scoring function with the goal that the correct explanation will be ranked high. Thus, the user will only have to browse through a few CCS to find the right set of causes. Obviously, the usefulness of such a ranking will depend directly on the scoring function. We argue that low numbers of side effects (not invalidating too many additional target attribute values) and small numbers of causes (Occam's razor) are good candidates for scoring functions (this will be shown experimentally in Section 7). In addition to reducing the load on the user, ranking can be exploited to avoid the exponential complexity of enumerating all possible CCS for an error set. However, even producing only the top-1 ranked CCS using these scoring functions is already NP-hard. Nonetheless, using a bottom-up generation approach with pruning and by partitioning the problem into smaller independent subproblems, we can reduce the runtime significantly (though obviously not avoid the theoretical exponential worst-case behaviour for each subset). We show that the complexity of combining the individual scores computed for each partition into a global score for the complete error set is in PTIME. Thus, partitioning is very effective for realistic DES, since in real schemas, the size of partitions tend to be small.

5.1 Ranking with Scoring Functions

We model ranking as sorting of cause sets based on scoring functions where lower scores are considered better.

DEFINITION 8 (SCORING FUNCTION). *A scoring function $f : \Lambda \rightarrow \mathbb{N}^0$ maps cause sets to natural number scores with $f(\varnothing) = 0$. A scoring function is called* ***weakly monotone*** *iff for all cause sets Λ and single causes λ:*

$$f(\Lambda) \leq f(\Lambda \cup \{\lambda\}) \leq f(\Lambda) + f(\{\lambda\})$$

A scoring function is called ***monotone*** *iff it is weakly monotone and for all cause sets Λ_a, Λ_b, Λ_c, and Λ_d:*

$$f(\Lambda_a) \leq f(\Lambda_b) \wedge f(\Lambda_c) \leq f(\Lambda_d) \Rightarrow f(\Lambda_a \cup \Lambda_c) \leq f(\Lambda_b \cup \Lambda_d)$$

$\lambda_1 = (\mathcal{T}_{SC}, \{E.e_2.Name\})$ $\quad$ $\text{Cover}(\lambda_1) = \{P.p_2.Name, O.o_2.Subordinate, O.o_7.Subordinate\}$

$\lambda_2 = (\mathcal{T}_{C}, \{C_1\})$ $\quad$ $\text{Cover}(\lambda_2) = \{P.p_1.Name, P.p_2.Name, P.p_3.Name\}$

$\lambda_3 = (\mathcal{T}_{SM}, \{\sigma_1\})$ $\quad$ $\text{Cover}(\lambda_3) = \{P.t'.A' \mid t' \in \{p_1, \ldots, p_5\} \wedge A' \in \{Name, HomeCity, Age\}\}$

$\lambda_4 = (\mathcal{T}_{JD}, \{F.f_2.FName\})$ $\quad$ $\text{Cover}(\lambda_4) = \{P.p_2.Name, P.p_3.Name, O.o_2.Subordinate, O.o_7.Subordinate\}$

$\lambda_5 = (\mathcal{T}_{JD}, \{A.a_2.Id\})$ $\quad$ $\text{Cover}(\lambda_5) = \{P.p_2.Name, P.p_3.Name, \}$

$\lambda_6 = (\mathcal{T}_{JM}, \{(F, c)\})$ $\quad$ $\text{Cover}(\lambda_6) = \{P.t.Name \mid t \in \{p_1, \ldots, p_3\}\}$

$\lambda_7 = (\mathcal{T}_{JM}, \{(A, e)\})$ $\quad$ $\text{Cover}(\lambda_7) = \{P.t.Name \mid t \in \{p_1, \ldots, p_3\}\}$

Figure 3: Causes for Error $P.p_2.Name$

*A scoring function is called **strongly monotone** iff it is montone and for all cause sets Λ_a and Λ_b with $\Lambda_a \cap \Lambda_b = \emptyset$:*

$$f(\Lambda_a \cup \Lambda_b) = f(\Lambda_a) + f(\Lambda_b)$$

5.2 Scoring on Side-effect and Explanation Size

We define two scoring functions that are of particular interest for ranking causes. Scoring function f_{Size} maps a cause set to its size ($f_{Size}(\Lambda) = |\Lambda|$) and function f_{SE} maps a cause set to its side-effect size according to E ($f_{SE}(\Lambda) = |\text{SE}(\Lambda, E)|$). A challenge for our approach is that these interesting scoring functions are not monotone, because of the potential overlap between cause sets and side effects.

Example 13. *Consider the CCS for two errors e_1 and e_2 with cause sets $\{\lambda_{1a}, \lambda_{1b}\}$ and $\{\lambda_{2a}, \lambda_{2b}\}$. Assume the side effects for these causes are as follows:*

$$\text{SE}(\lambda_{1a}) = \{R.t_1.a, R.t_2.a\}$$
$$\text{SE}(\lambda_{2a}) = \{R.t_3.a, R.t_4.a\}$$
$$\text{SE}(\lambda_{1b}) = \text{SE}(\lambda_{2b}) = \{R.t_1.b, R.t_2.b, R.t_3.b\}$$

Monotonicity is violated as $f_{SE}(\{\lambda_{1a}\}) \leq f_{SE}(\{\lambda_{1b}\})$ and $f_{SE}(\{\lambda_{2a}\}) \leq f_{SE}(\{\lambda_{2b}\})$, but $f_{SE}(\{\lambda_{1a} \cup \lambda_{2a}\}) \geq f_{SE}(\{\lambda_{1b} \cup \lambda_{2b}\})$. Similarly, for $\Lambda_A = \{\lambda_{1a}\}$, $\Lambda_B = \{\lambda_{2a}\}$, $\Lambda_C = \{\lambda_{1b}\}$, and $\Lambda_D = \{\lambda_{2a}\}$ then $f_{Size}(\Lambda_A) \leq f_{Size}(\Lambda_B)$ and $f_{Size}(\Lambda_C) \leq f_{Size}(\Lambda_D)$, but $f_{Size}(\Lambda_A \cup \Lambda_C) > f_{Size}(\Lambda_B \cup \Lambda_D)$.

This example demonstrates that these scoring functions are not monotone. However, they are weakly monotone.

Proposition 5.1 (Monotonicity of f_{Size} and f_{SE}). *Scoring functions f_{SE} and f_{Size} are weakly monotone, but neither monotone nor strongly monotone.*

Producing the top-k ranked CCS according to an arbitrary scoring function is hard in general, because the CCS enumeration is of exponential size in E. However, if the scoring function enjoys some monotonicity properties, we can exploit this to omit scoring (and generating) CCS if we can determine that they will not be beyond the top-k answers. Note that ranking CCS is quite different from top-k query processing, because we have to explore all possible CCS (basically a cross-product of the causes for each error grounding in E), while the goal of top-k query processing is to rank tuples in a relation and this relation usually contains only a small fraction of the cross-product of its attribute domains. Thus, top-k query processing approaches that sort the whole relation are not directly applicable to our problem. In addition, most top-k ranking algorithms, e.g., Fagin's algorithm [16], rely on scoring functions that, translated into our framework, are monotone. We show that generating the top-k causes according to our weakly monotone scoring functions f_{SE} and f_{Size} is NP-hard.

Theorem 1 (CCS Ranking Complexity). *Let E be an error set. Generating the top-k ranked CCS from enumeration $CCS(E)$ according to scoring functions f_{SE} and f_{Size} is NP-hard in the size of E even for $k = 1$.*

5.3 Combining Scoring Functions

5.3.1 Linear Combinations

Given a set of scoring functions that each evaluate different aspects of a cause set (e.g., its side-effect size or number of causes in the set as explained above) we may want to rank solutions by combining multiple aspects. Linear combinations of scoring functions and how monotonicity properties of scoring functions extend to linear combinations has been explored in [35]. Here give a short overview of these results.

A weighted combined scoring function $f_{WCS[g,w]}$ combines a finite number of scoring functions and assigns different weights w to each functions g. These weights can be either inputed by experienced data exchange users who can utilize their experience to determine reasonable bias by giving larger weights to more likely cause types, or learned from a set of ground truth CCS for a set of scenarios. Users may leverage their knowledge to create a suitable weighted combination for their type of specific data exchange scenarios.

Definition 9 (Linear Combination $f_{WCS[\vec{g},\vec{w}]}$). *Let $\vec{g} = g_1, \ldots, g_n$ be a vector of scoring functions, w_i with $w_i \in [0, 1]$ and $\sum_{i=1}^{n} w_i = 1$ be a vector of weights assigned to scoring functions in $\vec{g}$. The* Weighted Combined Scoring Function *$f_{WCS[\vec{g},\vec{w}]}$ is defined as:*

$$f_{WCS[\vec{g},\vec{w}]}(\Lambda) = \sum_{i=1}^{n} w_i * g_i(\Lambda) \quad (1)$$

We prove now that the linear combination of weakly monotonic scoring functions is also weak monotonic. This property makes sure that we can our incremental ranking algorithms introduced the following to rank according to this function.

Theorem 2 (Weak Monotonicity of $f_{WCS[\vec{g},\vec{w}]}$). *Let $\vec{g}$ be a vector of weakly monotone scoring functions. Any weighted combined scoring functions $f_{WCS[\vec{g},\vec{w}]}$ for $\vec{g}$ is weakly monotone.*

5.4 Incremental Ranking with Pruning

Despite the NP-hardness of the problem, we can nonetheless attempt to reduce the number of CCS that have to

Algorithm 1 ExplPruneRank

```
1: procedure ExplPruneRank(E,k, f, 𝕄)
2:     Q ← SortOnLower(Λ_1)
3:     while k > 0 do
4:         Cand ← popMin(Q)
5:         L ← length(Cand)
6:         if L = n then
7:             return Cand
8:             k ← k − 1
9:         else
10:            for all λ ∈ Λ_{L+1} do
11:                NewCand ← Cand ∪ {λ}
12:                computeScore(NewCand)
13:                insert(Q, NewCand)
```

be scored to produce the top-k answers. We create CCS bottom-up starting with singleton cause sets and use upper and lower bounds on the score of incomplete cause sets to prune partial solutions. Our ranking algorithm is a variant of A^*-search. Let $E = \{e_1, \ldots, e_n\}$ be an error set and Λ_i denote the set of causes for error e_i. For simplicity, we assume that each error has a single grounding, but the results directly translate to errors with multiple groundings by considering a set of error-grounding pairs instead of E.

Assume we are enumerating CCS by creating all possible CCS for $\{e_1\}$, then extending each of these sets to all possible CCS for $\{e_1, e_2\}$, and continue this process until all CCS for E have been generated. For any cause set Λ covering $\{e_1, \ldots, e_i\}$ generated by this process, the score of all extensions of Λ to a CCS for E according to a scoring function $f \in \{f_{SE}, f_{Size}\}$ can be bound by:

$$\textsc{Upper}(\Lambda) = f(\Lambda) + \sum_{k=i+1}^{n} \max_{\lambda \in \Lambda_k}(f(\{\lambda\}))$$

$$\textsc{Lower}(\Lambda) = \max(f(\Lambda), \max_{k \in \{i+1,\ldots,n\}}(\min_{\lambda \in \Lambda_k}(f(\{\lambda\}))))$$

These hold for any weakly monotone scoring function.

Lemma 5.1. *Let f be a weakly monotone scoring function and Λ a cause set explaining some errors in E. The following holds for every CCS Λ' that is an extension of Λ.*

$$\textsc{Lower}(\Lambda) \leq f(\Lambda') \leq \textsc{Upper}(\Lambda)$$

Our ranking algorithm (Algorithm 1) exploits these bounds. We keep a priority queue of partial solutions that is sorted on the lower score bound. This queue is initialized with a singleton set for each cause in cause set Λ_1 (the causes for error e_1). Afterwards, we pop the cause set Λ_{cand} with the smallest lower bound from the queue. Let this set be of size i. We then generate all possible extensions of Λ_{cand} with causes from Λ_{i+1} and insert them into the queue. If i is equal to n, then Λ_{cand} is the next top answer and should be returned. We iterate until k answers have been generated.

Theorem 3. *If f is a weakly monotone scoring function, then Algorithm 1 returns the top-k ranked causes for input error set E according to f.*

5.5 Leveraging Partitioning

As shown in the last subsection, the cause ranking problem according to scoring functions f_{Size} and f_{SE} is NP-hard in the number of errors. The main reason for the complexity of ranking is the potential overlap between the coverage of causes (and their side effects). Because of this overlap the scoring functions are only weakly monotonic. However, for typical data exchange scenarios, the causes for two errors will not overlap if the mappings that generated these error have no overlap in relations, correspondences, and transformations. For instance, consider tgds σ_2 and σ_3 from the running example in Figure 1. These tgds do not overlap. Hence, neither causes nor side effects for errors in tuples generated by these tgds will overlap. Our next contribution is to exploit this independence to make ranking more efficient.

Definition 10 (Independent Partitioning). *Let E be an error set. A partitioning $P = \{P_1, \ldots, P_n\}$ of E into disjoint partitions is called an independent partitioning if for any two partitions P_i and P_j of P and any CCS Λ_i and Λ_j for these partitions their causes and coverage are disjoint.*

$$\forall P_i, P_j \in P : \forall \Lambda_i \in CCS(P_i), \Lambda_j \in CCS(P_j) : \Lambda_i \cap \Lambda_j = \varnothing \wedge \textsc{Cover}(\Lambda_i) \cap \textsc{Cover}(\Lambda_j) = \varnothing$$

Based on independent partitioning we define a monotonicity property of scoring functions according to unions of cause sets from different partitions. This property will be used to efficiently combine CCS rankings for individual partitions into a global ranking for an error set E.

Definition 11 (Partition Monotonicity). *Let E be an error set and P an independent partitioning of E. A scoring function f is strongly partition monotone iff:*

$$\forall P_i, P_j \in P : \Lambda_i \in CCS(P_i), \Lambda_j \in CCS(P_j) : f(\Lambda_i \cup \Lambda_j) = f(\Lambda_i) + f(\Lambda_j)$$

The two scoring functions f_{SE} and f_{Size} are strongly partition monotone. Thus, we can apply the efficient ranking algorithm presented below as long as we are able to find an independent partitioning.

Proposition 5.2. *Scoring functions f_{SE} and f_{Size} are strongly partition monotone.*

Given this result, we can apply the efficient ranking algorithm presented below as long as we are able to find an independent partitioning.

We now introduce the *scenario graph*, a representation of a DES which we use to generate an independent partitioning.

Definition 12 (Scenario Graph). *Let $\mathbb{M}$ be a data exchange scenario. The scenario graph $SG(\mathbb{M})$ is a graph (V, E) where V contains a vertex for every source or target relation attribute and one vertex for each variable used in a mapping. There exists an edge between a source attribute $R.A$ and a mapping variable $\sigma.V$ if $\sigma.V \in var(R.A)$. There exists an edge between $\sigma.V$ and target attribute $R.A$ if $\sigma.V \in var(R.A)$. Two mapping variable vertices are connected if they belong to the same mapping: $(\sigma.V, \sigma'.V') \in E$ iff $\sigma = \sigma'$.*

Let E be an error set for a DES $\mathbb{M}$. A scenario graph partitioning of E places errors $R.t.A$ and $R'.t'.A'$ in the same partition if their $R.A$ and $R'.A'$ are in the same connected component of the scenario graph $SG(\mathbb{M})$.

The scenario graph of a mapping scenario concisely captures potential overlap between causes and side-effect at the schema level.

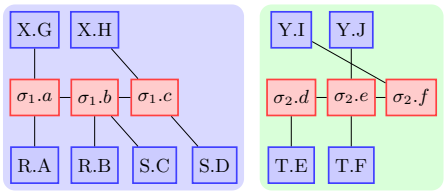


Figure 4: Example Scenario Graph

EXAMPLE 14. *Consider a scenario with two mappings* $\sigma_1 : R(a,b) \wedge S(b,c) \rightarrow X(a,c)$ *and* $\sigma_2 : T(d,e) \rightarrow Y(f,e)$ *over relations* $R(A,B)$, $S(C,D)$, $T(E,F)$, $X(G,H)$ *and* $Y(I,J)$. *Figure 4 shows the scenario graph for this scenario. For example, target attributes X.G and Y.I are in different components and, thus, the coverage and side-effects of causes for errors from these two attribute cannot overlap.*

The partitioning of an error set based on the scenario graph is independent. However, we are being conservative in using the connected components of the graph.

THEOREM 4. *Let* E *be an error set for a DES* $\mathbb{M}$. *The scenario graph partitioning* $P_{\mathbb{M}}$ *of* E *is independent.*

Using an independent partitioning, we can partition an error set E, rank each partition individually, and combine the rankings into a global ranking. The main concern here is how to create a global ranking without the need to fully materialize the exponential size ranking for each partition. The algorithm we present next is based on the following observation. Consider a partitioning $P_{\mathbb{M}} = \{P_1, \ldots, P_n\}$. We use $\Lambda_i[j]$ to denote the CCS of rank j for E_i according to the scoring function. We can represent a CCS Λ for the complete error set E as vectors v_Λ of ranks for the per-partition CCS that were used to construct the global CCS. For example, if we have two partitions, then $[1,2]$ represents the CCS built from the 1^{st} ranked CCS for partition P_1 and the 2^{nd} ranked CCS for partition P_2. Given that partition monotonicity holds, we know that $\forall i \in \{1,\ldots,n\} : v_\Lambda[i] \leq v_{\Lambda'}[i]$ for two global CCS Λ and Λ', iff $f(\Lambda) \leq f(\Lambda')$. For example, there can be no better CCS than $[1,\ldots,1]$. The algorithm *PartitionRank* (Algorithm 2) is based on this observation. It first creates a scenario graph partitioning $P_{\mathbb{M}}$ of the input error set E (line 2). For each partition it initializes a ranker using a variant of Algorithm 1 that supports incremental access to the produced ranking. If the n^{th} ranked solution is requested, this variant will produce the top-n CCS and keep its state between calls to service future requests. PartitionRank uses two main data structures: 1) a list Q of CCS vectors sorted on score (implemented as a tree data structure with $O(log(n))$ look-up and insert) and 2) a set *Done* of CCS vectors ($O(1)$ test whether a vector has been produced before). At each point in time a prefix of Q represents the partial ranking produced so far. Variable *ranked* stores the length of this prefix. Both Q and D are initialized with $[1,\ldots,1]$ (line 4). The algorithm repeats the following procedure until *ranked* is equal to k, i.e., it has produced the top-k answers. We retrieve the k^{th} vector v_k from Q (line 6), then we create n variations of v_k by increasing each position in the vector by one (lines 9-10). For each variation that has not been produced before (line 11), we compute its score by summing up the individual scores for each partition CCS and then insert it into Q and *Done* (lines 12-15).

Algorithm 2 PartitionRank

```
1: procedure PartitionRank(E,k, f, 𝕄)
2:     P_𝕄 ← SGpartition(E, 𝕄)
3:     n = ‖P_𝕄‖
4:     Q ←< [1,...,1] >, Done ←< [1,...,1] >, ranked ← 1
5:     while ranked < k do
6:         Cur ← getElem(Q, ranked)
7:         for i ∈ {1,...,n} do
8:             if Cur[i] < ‖CCSEP_i‖ then
9:                 New ← Cur
10:                New[i] ← New[i] + 1
11:                if New ∉ Done then
12:                    computeScore(New)
13:                    insert(Q,New)
14:                    insert(Done,New)
15:        ranked ← ranked + 1
```

Afterwards, we increase *ranked*. Set *Done* is used to avoid reinserting a vector more than once. The access to actual CCS represented by the vector positions is only needed to produce the final result CCS and to get the scores of $\Lambda_i[j]$ when inserting a new vector into Q.

THEOREM 5 (CORRECTNESS). *Let* P *be an independent partitioning of an error set* E *and* f *a partition monotone scoring function. Algorithm PartitionRank returns the top-k answers for* E *according to* f.

As proven in the following theorem, combining per-partition rankings into a global top-k ranking using Algorithm *PartitionRank* is efficient, in $O(k \cdot n \cdot log(k \cdot n))$. Furthermore, we need to access at most the top-k ranked solutions for each per-partition ranker.

THEOREM 6 (COMPLEXITY). *Given an independent partitioning* P *of an error set* E *and an top-k ranking for each partition according to a strongly partition monotone scoring function* f, *Algorithm PartitionRank generates a top-k ranking for* E *in* $O(k \cdot \|P\| \cdot log(k \cdot \|P\|))$.

THEOREM 7 (PER-PARTITION BOUNDS). *Algorithm PartitionRank at most accesses the top-k per-partition CCS to produce the global top-k CCS.*

5.6 Combining Multiple Score Functions Using Skyline Ranking

The algorithms presented in the previous subsections enable us to rank CES based on a single ranking function. However, what if we want to rank on multiple criteria, e.g., small size and small number of side-effects? One solution is to define new scoring functions which are weighted combinations of scoring functions. For example, we could define $f' = w_1 \cdot f_{Size} + w_2 \cdot f_{SE}$. As long as the weights are positive and all involved scoring functions are weakly monotone, then f' is also weakly monotone and, thus, we can still apply algorithm *ExplPruneRank*.

THEOREM 8 (COMBINING SCORING FUNCTIONS). *Let* $f_1, \ldots, f_n$ *be weakly monotone scoring functions and* $w_1, \ldots, w_n$ *be a list of values from* $[0,1]$. *The scoring function* f' *defined as* $f'(\Lambda) = \sum_{i=1}^{n} w_i \cdot f_i(\Lambda)$ *is weakly monotone.*

PROOF. We have to proof that

$$f'(\Lambda) \leq f'(\Lambda \cup \{\lambda\}) \leq f'(\Lambda) + f'(\{\lambda\}) \qquad (2)$$

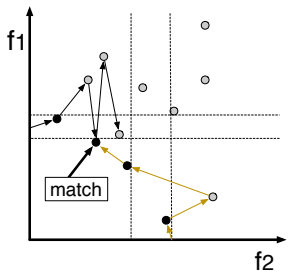


Figure 5: Example Skyline Computation

given that

$$f_i(\Lambda) \leq f_i(\Lambda \cup \{\lambda\}) \leq f_i(\Lambda) + f_i(\{\lambda\}) \quad (3)$$

for each $i \in \{1, \ldots, n\}$. Multiplying inequality 3 with a positive weight w_i does result in

$$w_i \cdot f_i(\Lambda) \leq w_i \cdot f_i(\Lambda \cup \{\lambda\}) \leq w_i \cdot (f_i(\Lambda) + f_i(\{\lambda\})) \quad (4)$$

Since each element in the inequality is positive we can sum up the inequalities for all $i \in \{1, \ldots, n\}$ resulting in inequality 2. □

The problem with this approach is that it is hard to determine what are good choices for the weights. Alternatively, we could use the well-known skyline operator to return all solutions where there is no provably better solution. Given a set P of points in a n-dimensional space, the skyline operator returns all points that are not dominated by any other point. A point p dominates a point p' written as $p \prec p'$ if p is at least as "good" as p' in every dimension and "better" in at least one dimension:

$$\forall i \in \{1, \ldots, n\} : p[i] \leq p'[i] \wedge \exists i \in \{1, \ldots, n\} : p[i] < p'[i]$$

For explanation ranking we represent CES as points by mapping them onto their scores according to a set of scoring functions $F = \{f_1, \ldots, f_n\}$:

$$\Lambda \rightarrow [f_1, \ldots, f_n]$$

For example, if we compute the skyline according to f_{Size} and f_{SE}, then for every returned CES there exists no CES which has lower or equal scores in both f_{Size} and f_{SE}, and a lower score in either f_{Size} or f_{SE}. From now on we use Λ to refer an explanation set or its vector of scores.

The skyline operator saves us from having to come up with reasonable weights. However, typical implementations of this operator are not suitable for our problem, because they would require us to materialize all CES for an error set E to compute the skyline. We now discuss adaptations of skyline algorithms which use incremental rankers for the scoring functions over which the skyline should be computed. Thus, our adaptations are applicable if F contains only weakly monotone scoring functions.

Many different algorithms for the skyline operator have been introduced in related work (e.g., see [14]). We try to avoid materializing a complete CES enumeration during ranking which naturally disqualifies algorithms that index the whole dataset or are not progressive (do output points in the skyline before having processed the whole dataset). Even though we cannot sort the whole dataset, because this would require materialization of the CES enumeration, we can still access CES ordered according to any weakly monotone scoring function from F by using Algorithm *ExplPruneRank*. We now briefly review two well-known skyline algorithms and discuss how they can be adapted for CES ranking.

B-tree algorithm: The B-tree algorithm from [9] assumes that there exists a B-tree index for each dimension. For CES ranking that corresponds to creating one index on $f_i(\Lambda)$ per $f_i \in F$ which stores a complete CES enumeration. These B-tree indexes are traversed simultaneously starting from the lowest value until there exists a point p that has been visited in all indexes. Once such a point p has been found, it is guaranteed that have not been traversed will not be in the skyline. Instead of using B-trees, we can apply our ranking algorithm to traverse the CES sorted on a particular scoring function f_i. Figure 5 shows an example skyline computation for two dimensions. The solid black points are on the skyline. The paths starting from the x respective y axis indicate the order of traversal using the rankers for f_1 and f_2. Once the first match has been found, any standard skyline algorithm can be used to check whether the points we have traversed so far are in the skyline or not. The advantage of this algorithm is that it is trivial to implement on top of *ExplPruneRank*. However, depending on the data distribution we may have to traverse most CES before the first match is found.

SalSa: The SalSa algorithm [6], and extension of the block nested loop algorithm [9] sorts the input on a monotone sorting function $\mathcal{M}$ (here monotone means $\mathcal{M}(\Lambda) > \mathcal{M}(\Lambda') \Rightarrow \Lambda \not\prec \Lambda'$) and then loops through the points in sort order. The algorithm can terminate once a stop criterion is met. This criterion depends on the choice of $\mathcal{M}$. The sort order proposed in [6] (translated into our terminology) sorts on $\mathcal{M}_1(\Lambda) = min_{i \in \{1,\ldots,n\}}(f_i(\Lambda)))$ and uses $\sum_{i \in \{1,\ldots,n\}}(f_i(\Lambda))$ as a secondary sort criterion to disambiguate points with the same $\mathcal{M}_1$ value. We can sort points according to this order, by traversing rankers for all scoring functions in F simultaneously. We keep a priority queue of (i, Λ) pairs sorted on $\mathcal{M}$. Initially we insert the first element from each ranker into this queue. To get the next element in sort order we pop the first element (i, Λ) from the queue and then insert the next element from the ranker for f_i (if it exists) into the queue. In addition we keep track of the CES we have seen so far to avoid inserting a new CES if it was already returned previously by a different ranker. The only difference to the original SalSa algorithm is the incremental sorting using the rankers.

Algorithm for 2D Skylines: If F contains only two scoring functions, we can optimize the B-tree algorithm as follows. We keep a current upper bound $u_{1/2}$ for each of the two dimensions. Upper bound u_1 is updated for f_1 to $f_1(\Lambda)$ whenever we retrieve the next CES Λ from the ranker for scoring function f_2 and vice versa. Once we retrieve an explanation set Λ from the ranker for f_1 with $f_2(\Lambda) \geq u_2$ the algorithm terminates. The oposite holds for explanation sets from the ranker for f_2. This algorithm is shown in Algorithm 3. This algorithm uses the incremental version of Algorithm *ExplPruneRank* which preserves its state between calls. Using the upper bound, we can determine during the initial traversal of the rankers which points belong to the skyline instead of having to compute the actual skyline from the points traversed before a matching point was found.

Algorithm 3 Expl2DSkyLine

```
1: procedure ExplSkyLine(E,k, f1, f2, M)
2:     k1 = 1, k2 = 1, u1 = ∞, u2 = ∞
3:     Λl = NULL, sky = ∅
4:     while true do
5:         for all d ∈ {1, 2} do
6:             o ← 3 − d
7:             Λ ← ExplPruneRank(E, kd, fd, M)
8:             if f1(Λ) < u1 ∧ f2(Λ) < u2 then
9:                 Λl ← Λ
10:                uo ← fo(Λ)
11:                sky ← sky ∪ {Λ}
12:            else if fd(Λ) = fd(Λl) ∧ fo(Λ) = fo(Λl) then
13:                sky ← sky ∪ {Λ}
14:            else if fd > ud then
15:                return sky
16:            kd ← kd + 1
```

Once the skyline is computed we may still want to order the points in the skyline based on some preference. For example, the user specifies that the results should be ordered on one of the scoring functions in F. This type of ranking has been called ranked skyline queries in the literature [14]. We can use e.g., Fagin's algorithm to implement this type of ranking on top of the skyline results as long as the ranking function is monotone in f_i values.[3]

5.7 Discussion

We have introduced an approach for incrementally ranking causes using on scoring functions. The approach prunes the search space based on a lower bound proven for weakly monotone scoring functions. Ranking is NP-hard for the scoring functions discussed in the paper. By partitioning a ranking problem into independent subproblems based on mapping scenario information we are able to reduce the problem size. Algorithm *PartitionRank* enables us to efficiently combine per-partition rankings into a global ranking.

6. IMPLEMENTATION

6.1 Overview

We now give an overview of the implementation of our cause generation and ranking techniques in Vagabond. Our system consists of three components: 1) a GUI for marking errors and browsing ranked causes, 2) a middleware implementing our cause generation and ranking algorithms, and 3) a TRAMP database server used for storing instance data, DES information, and generating provenance.

6.1.1 GUI

The user can interact with the system through a GUI [22]) that allows the user to browse the data to mark errors and inspect all elements (e.g., mappings, correspondences) of a DES. The GUI permits the visual specification of queries that determine which cells should be marked as errors Expert users can write these queries directly whereas less experienced users can use the GUI to specify query conditions (e.g., mark all names of persons that live in Toronto).

6.1.2 Cause Engine

We have implemented cause generation and ranking in a Java middleware. Given a set of errors E and scoring function f, the system generates all groundings (Section 6.2) and computes all causes for each grounding individually. (Section 6.3). The result is a set of potential causes for each grounding. We then proceed by constructing a scenario graph, partition the error set (and cause set) based on this graph, We compute a partition, initialize rankers for each partition, and create a global ranker using Algorithm 2. (Section 6.5). The engine exposes the ranking through an iterator interface used by the GUI to enable interactive exploration of the ranking.

6.1.3 Backend Provenance Computation

Computing causes requires the computation of several forms of provenance. Vagabond uses a *TRAMP* [21] server for storing DES information, instance data, and accessing provenance. Alternatively, we could use the approach pioneered in the Orchestra system [24] of adding functions to provenance polynomials to encode mapping application. Both mapping and influence provenance can be extracted from this representation. With either backend, for efficiency, we need to do some optimization to avoid redundant computation of provenance. These optimization are described below.

6.1.4 TRAMP

Given an error e we how can we effectively determine all potential causes for each grounding of e and their side-effects (coverage) according to the cause types introduced in Section 4.2? It turns out that the key is efficient provenance computation and access to DES elements. We use a *TRAMP* [21] server for storing DES information, instance data, and accessing provenance. TRAMP is an extends PostgreSQL with support for generating provenance and querying provenance and DES information. TRAMP implements the "use SQL to generate and query the provenance of SQL queries" paradigm pioneered by Perm [20]. Using the SQL extensions of the system, the user can retrieve the provenance of all result tuples of a query in a relational encoding. This representation pairs query result tuples with provenance information. Unless requested provenance is not materialized, but reconstructed on the fly using query rewrite techniques. For details of the approach please see [21, 20]. TRAMP supports the three provenance types introduced in Section 3.0.3. The cause generation of Vagabond uses the query interface of TRAMP to 1) compute all error groundings, 2) retrieve various types of provenance during cause generation, 3) and query DES information.

6.2 Generating Error Groundings

For each error e in the input error set E we determine its groundings $\Theta(e)$ as follows. The mapping provenance of e contains all tgds to be grounded and the influence provenance, by modelling from which input tuples a tuple is derived, provides the tuples to be used in the grounded atoms.[4]

[3]Note that this does not imply that the scoring functions in F have to be monotone as defined in Section 5.1, because the ranking function is applied to the skyline results is evaluated over the vector representation $[f_1(\Lambda), \ldots, f_n(\Lambda)]$ of an explanation set Λ.

[4]Alternatively, we could use the approach pioneered in the Orchestra system [24] of adding functions to provenance polynomials to encode mapping application. Both mapping and influence provenance can be extracted from this repre-

Both types of provenance can be retrieved by issuing a provenance query using TRAMP. We build a join graph skeleton for each tgd σ of the data exchange scenario storing the structure of the join graph. Thus, we store the structure of the join graph for σ only once and encode join graphs for individual groundings of σ as lists of tuples.

6.3 Cause Generation

We now describe how to, given an input error e and one grounding $\sigma[\theta] \in \Theta(e)$, generate all causes of a certain type by accessing the data exchange scenario and provenance using a TRAMP server instance. Because of space limitations, we only discuss one cause type here.

6.3.1 Copy Data Error

Given an error marker $e = R.t.A$ and grounding $\sigma[\theta]$ we can determine the single copy data error λ for this combination by computing the copy provenance of e restricted to the grounding $\sigma[\theta]$ (recall that $O_I = \mathcal{C}(e, \sigma[\theta])$ for this type error). Using TRAMP this information is retrieved by executing a selection over a query generating the copy provenance of relation R from $R.t.A$. To compute the coverage of λ, we need to find all target attribute values e' which have been copied from any $e \in O_I$. We first partition the set O_I into subsets containing only attribute values from a single source relation. This is necessary, because TRAMP only supports generating the copy provenance of a single target relation at once. For each relation R' that has a corresponding partition $P_{R'}$ we determine to which target relation attributes the affected attributes are copied too by analyzing the mappings. For each affected target relation T we check wether any of the errors from source relations potentially affecting an attribute value in relation T are in the copy provenance of a target attribute. In TRAMP this can be expressed by retrieving the copy provenance of all tuples in T and checking whether the provenance overlaps with O_I. This operations can be executed efficiently using TRAMP, because the selection will usually be pushed into the provenance computation by the system's optimizer. Nonetheless, further performance improvements are certainly possible. For instance, we could index copy provenance using the index structure from Kementsietsidis et al. [25].

6.3.2 Join Data Error

For generating all join data error explanations for an error e and grounding $\sigma[\theta]$ we use the join graph generated for $\sigma[\theta]$ generated during the computation of $\Theta(e)$. We then determine the set of potentially incorrect join attributes from the join graph as explained in the previous section. Since the structure of the join graph is fixed for a given σ, we only generate the general structure once and represent a particular instance of the join graph for a grounding $\sigma[\theta]$ as a list of tuple identifiers. Note that the helper structures $\text{PATH}(R(t), \sigma[\theta])$ (path in the join graph that start in the node representing the erroneous tuple), $\text{PVARS}(e, \sigma[\theta])$ (variables in the grounding the may cause incorrect joins), and $O_I(R(t), v)$ (source attribute values that where used in the joins of the grounding) are in a sense independent of the actual tuples in the grounding. For example, the relations on paths in $\text{PATH}(R(t), \sigma[\theta])$ for different t and θ only differ in the grounded tuples, but not in the relations. Thus, we can generate template versions of $\text{PVARS}(R.t.A, \sigma[\theta])$ and $O_I(R'(t'), v)$ upfront where t, t', and θ are parameters for these templates. We then generate all join data errors by producing one error for each atom-variable combination $(R'(t'), v)$ on a path in $\text{PVARS}(e, \sigma[\theta]$ by instantiating the template with tuples from the grounding $\sigma[\theta]$.

To compute the coverage for one such join data error $\lambda_{(R'(t'),v)}$ we need to find all groundings where any for tgds σ' where the attribute values in O_I have been used in a join and determine which target attribute values would be effected by this join. We first need to determine all other atoms R'' for which there exist a grounding $\sigma'[\theta']$ so that $(R'(t'), v')$ is on a path in $\text{PATH}(R''(t''), G(\sigma[\theta']))$ and $attr(R'(t'), v) = attr(R'(t'), v')$. Again we build a data structure that is independent of the tuples in the grounding. This data structure helps us to compute which potential atoms may appear in such groundings. Using the join graph skeletons of all tgds we determine for which atoms R'' of which tgd σ', (R', v') appear in a path in $\text{PATH}(R''(t''), G(\sigma'[\theta']))$. The data structure we are using is a map from atoms-variable tuples to tgd-atom-attribute tuples that maps (R', v) to all (σ', R'', a'') for which $(R'(t'), v)$ is on a path in $\text{PATH}(R''(t''), G(\sigma'[\theta']))$. We partition these tgd-atoms-variable tuples according to the target relation T. For each target relation T we issue a query to determine all t'' for which there is a grounding $\sigma'[\theta']$ for which $R''(t'') \triangleleft \sigma'[\theta']$ and R', v in $\text{PATH}(R''(t''), G(\sigma'[\theta']))$. Each such query retrieves the influence provenance of tuples in T and selects only those $t'' \in T$ for which R' is contained in one witness list corresponding to a grounding of σ'. This is be implemented as a selection condition over the provenance.

6.3.3 Correspondence Error

To generate a correspondence error for $(e, \sigma[\theta])$ we need to know which correspondences are used by tgd σ to copy which attribute. Recall that we assume that the correlation between correspondences and mappings is provided with the data exchange scenario $\mathbb{M}$ (the mapping MAP). We retrieve these correspondences $\Gamma(\sigma)$ and check which ones are mapping to attribute $R.a$. Let Γ' denote the set of these dependencies. We generate a correspondence error $\lambda = (C, \{\Gamma', \Sigma(\Gamma')\})$ for these correspondences. To determine the coverage we determine which target relations are effected by tgds in $\Sigma(O_\Gamma)$, that is tgds that use a correspondence in O_Γ. For each of these target relations we construct a query that retrieves the tuples generated by one of the affected mappings using the mapping provenance. The correspondences are used to find out which attributes are affected for each returned tuple.

6.3.4 Superfluous Mapping Error

For this error type we assume that the tgd σ of grounding $\sigma[\theta]$ is superflous. To compute the coverage we determine which target relations are generated by this mapping (all relations mentioned in the RHS of the tgd). For each such target relation T we generate a query that retrieves the mapping provenance for each tuple t' in the relation and checks whether σ is in the mapping provenance. All attributes of each such tuple t' are part of the coverage.

6.3.5 Join Mapping Error

To find join mapping errors for an error marker e and grounding $\sigma[\theta]$, we use the template data structures for PVARS created for source join errors. We determine all pairs

sentation.

(R', v) in $\text{PATH}(e, G(\sigma)\theta)$ and determine the attributes on the incoming edge to each $R'(t')$. For each combination (R', v) we create a join mapping error. The coverage for such an error contains all target attribute values R'', t'', A'' for which R', v is on a path in $\text{PATH}(R''(t''), G(\sigma'[\theta']))$. The R'' and A'' part can be determined without accessing the data by analyzing the join graph templates. This analysis is done once and the results are preserved for future join mapping error computations. For each pair (R'', A'') we construct a query for retrieving all tuples produced by σ (mapping provenance). The A'' attribute of each such tuple belongs to the coverage of the explanation.

6.4 Marking Errors

In Vagabond, the user can inform the system about target instance errors by either marking attribute values through Vagabond's graphical user interface or by running queries that return errors to be marked. We call this type of queries *marker queries*. A marker query is essentially an SQL query that returns an relational encoding of an error set (the error relation introduced below). The user can either write such a query by hand or use a wizard provided by Vagabond.

6.5 Error Set Partitioning and Ranking

We first build a scenario graph $SG(\mathbb{M})$ from the input DES $\mathbb{M}$ in memory. This graph is then used to create an scenario graph partitioning of the input error set E and, thus, also of the causes we have generated in the previous step. We then initialize an incremental ranker (Algorithm 1) for each partition $P \in P_{\mathbb{M}}$ and a partition ranker (Algorithm 2) using the individual rankers. The user navigates the ranking using the controls of the GUI which in the background causes calls to the partition ranking algorithm to generate additional solutions if necessary. Recall that both the individual and the partition ranking algorithm are incremental, i.e., the queue data structures are kept between calls.

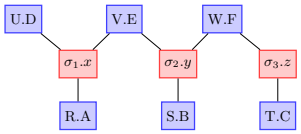
6.6 Optimizations

6.6.1 Avoiding redundant cause generation

As introduced in Section 6.3, generating causes for a set of errors requires execution of queries to retrieve various types of provenance parts of the target instance. Some types of causes (e.g., superfluous mapping errors) cover large number of target cells which results makes computing the coverage of such an cause expensive. Generating such causes multiple times - once for error that is covered by the cause - is unnecessary. Thus, for all cause types which typically have large coverage and low number of distinct causes per scenario we cache the causes to avoid generating them more than once. Currently, we have implemented this optimization for superfluous mapping errors, source skeleton, and correspondence errors.

6.6.2 Finer-grained partitioning

The scenario graph based partitioning we have introduced in Section 5.5 is quite conservative, i.e., two errors may be in the same component even if their causes and causes' side-effects cannot overlap. We improve partitioning by adapting the graph based on the input error set E. Note that causes and side-effects for errors e_1 and e_2 from two target attributes A_1 and A_2 can only overlap if the tgds mapping to A_1 and A_2 are directly connected (there is an edge between a variable of the tgd and a node) to at least one common source or target attribute. Obviously, this condition implies that A_1 and A_2 are in the same connected component of the scenario graph. However, the oposite does not hold necessarily. For instance, consider the example scenario graph shown below and assume we have two errors e_1 in U.D and e_2 in W.F. Target attributes U.D and W.F are in the same (single) connected component of the scenario graph. However, the causes and side-effects of e_1 and e_2 cannot overlap. Our approach for computing a more finer-grained partitioning based on this operation works as follows. We start from singleton components - one for each target attributes for which these is at least one error in E and grow subgraphs starting from these single nodes. We first determine all source relations that are connected through at least one mapping variable to a target attribute, then we extend this set of nodes by adding all attributes from partially covered source relations and tgds. Finally, we determine all target attributes directly connected to one of the nodes. Afterwards, we proceed by merging subgraphs that have at least one node in common into a bigger components. We repeat the later step until a fix point is reached. The result is again an independent partitioning which is typically smaller than a partitioning based on connected components and in worst-case degrades to a partitioning based on connected components.



6.6.3 Ordering Errors for Ranking

The performance of our incremental ranking technique depends mainly on how effectively we can prune the search space. The worst-case is if all causes for each error have very similar or even the same score, because then the lower score bounds will also be very similar which limits pruning opportunities. While we cannot control the scores of different causes we can control in which order the errors are accessed by the algorithm. As a heuristic, we sort the errors decreasing on the size of the interval spanned by the lowest score and highest score for causes for the error. The rationale is that this leads to more pruning opportunities because the partial solutions we create are likely to have very diverse scores which would allow us to prune some of them early on.

7. EXPERIMENTS

We evaluate (1) the performance of our techniques for scenarios of different size, structure, and complexity as well as (2) test the quality of the rankings produced by the algorithms introduced in Section 5. All experiments were run on a machine with one Intel i7-4510 2GHz CPU and 8GB RAM running Ubuntu 14.04 64-bit.

7.1 Scenario Generation and Setup

We use the *iBench* [5] integration task generator, to evaluate the quality of rankings and test performance over diverse and realistic data exchange scenarios. iBench creates the schemas, data, mappings, and correspondences of a data exchange scenario by applying and combining mapping primitives (that represent common transformation patterns like vertical partitioning or denormalization) into more complex

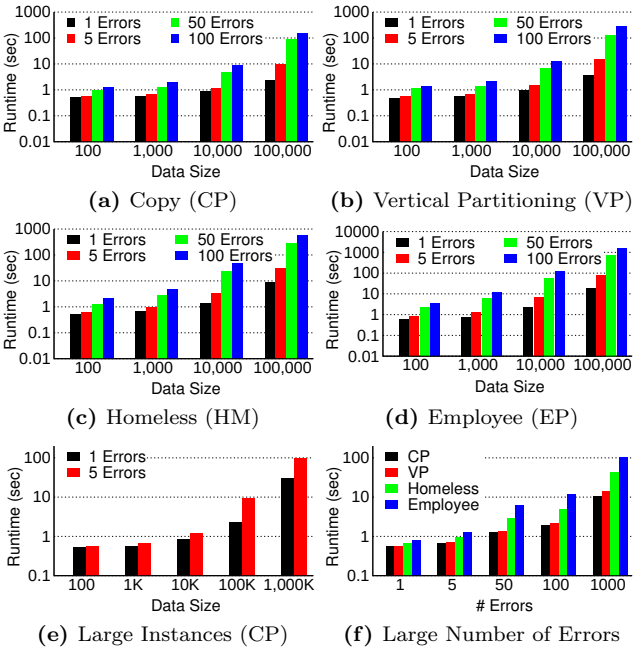


Figure 6: Number of Errors and Instance Size

mapping scenarios. iBench can produce all elements of data exchange scenarios as defined in Section 3. With iBench we can control the size of the data, along with both the size and complexity of the generated mappings and schemas. In addition, iBench permits direct control of the amount of sharing of source/target relations among the generated tgds. This feature permits us to control (and evaluate) the effect of partitioning on performance for scenarios with a realistic amount of sharing. iBench also allows the use of real scenarios and can scale them by increasing the data and schema size as well as combining them with mapping primitives.

We have generated several groups of scenarios in different data size, schema size, and mapping complexity (measured by number of atoms in tgds). The *Copy (CP)* scenarios consist only of copy primitives (source relations are copied to the target, though with sharing, the same source relation may be copied many times or multiple source relations may be copied to the same target). For the *Vertical Partition (VP)* scenarios, source relations are partitioned into multiple target relations (again mappings may share relations). The *Employee* scenario is an extended version of our running example. The *Homeless* scenario is the running example from TRAMP [22]. In addition, we consider a scenario based on a mapping between Schema 1 and Schema 3 of Amalgam [30]. Amalgam is an integration benchmark using real schemas storing bibliographic data.

7.2 Cause Generation Performance

7.2.1 *Number of errors and instance size*

We now evaluate the scalability of cause generation when varying the number of errors (1 to 100) and instance size (100 to 100K tuples per relation). The results for our four main scenario are shown in Figure 6. Here, we consider simple copy and vertical partitioning scenarios with one instance of this mapping primitive (only one source table copied to one target table in case of copy). Since causes are generated for each error individually, we would expect cause generation to scale linearly in the number of errors. Our results confirm this assumption, showing a sightly sublinear increase in runtime when increasing the number of errors explained by our optimization that caches causes which cover large number of target cells such as superfluous mapping or correspondence errors. Interestingly, cause generation also exhibits a less than linear growth in the size of the instance. Increasing the instance size by 3 orders of magnitude leads to a runtime increase of less than one order of magnitude for the tested scenarios. To evaluate whether this trend also exists for even larger instances we have scaled the copy scenario up to 1M tuples. The results are shown in Figure 6e. Again, we observe a linear growth of runtime in instance size. We also tested larger error sets (100K instance) to confirm that we still get linear or sublinear runtime increase. The results are shown in Figure 6f. Clearly, the runtime grows linearly as expected.

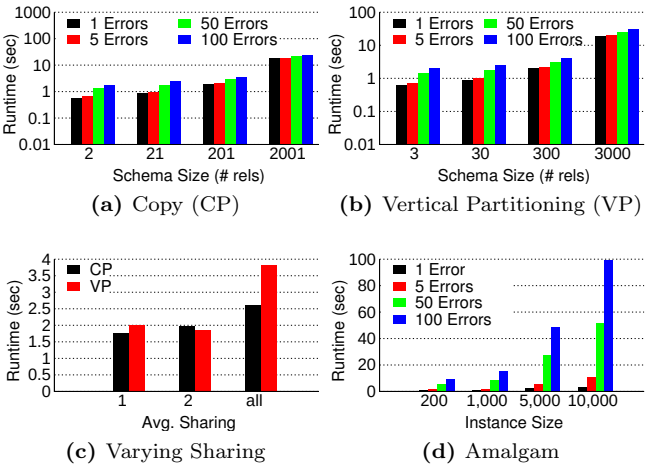
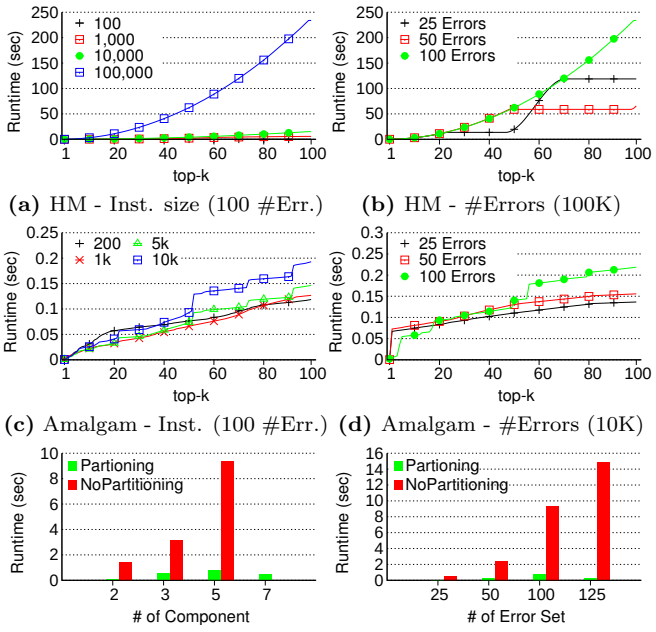


Figure 7: Schema Size and Amalgam

7.2.2 *Varying schema complexity and Amalgam*

Here, we fixed the instance size to 1K tuples per source relation and measure runtime of cause generation while varying the size of the input schemas. Figure 7a and 7b show the results for the CP and VP. The x-axis shows the number of relations in the schema. VP splits one source relation into 2 target relations and, thus, a schema size of 3000 corresponds to 1000 VP primitives. The results demonstrate the our approach can deal with realistically sized schemas. Increasing schema size by 4 orders of magnitude results in a moderate increase of 2 orders of magnitude in runtime. The complexity of a mapping scenario may also influence performance of cause generation. In this experiment, we fixed schema size and instance size and varied the amount of sharing for tgds, i.e., a high percentage of sharing of source schema elements means that a single source relation will be used in multiple tgds leading to more complex transformations and interdependencies among tgds and other scenario elements (e.g., correspondences). More complex scenarios typically lead to an increase in the number of causes, side effects, and an overlap between causes and their side effects. Figure 7c shows the results for the CP and VP with 100 primitives for increasing amount of sharing (reported as the



(a) HM - Inst. size (100 #Err.) **(b)** HM - #Errors (100K)

(c) Amalgam - Inst. (100 #Err.) **(d)** Amalgam - #Errors (10K)

(e) W. vs. W/O Part. - #Comp. **(f)** W. vs. W/O Part. - #Errors

Figure 8: Ranking Performance

average number of mappings that share a relation). To evaluate whether our system can deal with the intricacies of real world schemas we use the Amalgam scenario with instance size between 200 and 10k tuples (using iBench to scale the data). Figure 7d shows the results of this experiment. The results confirm cause generation scales well (linearly) in the number of errors and data size - even for real scenarios.

7.3 Ranking Performance

7.3.1 *Incremental Ranking*

We now measure the performance of incremental ranking and compared it against a naive top-k approach that generates and sorts all CCS upfront. Figures 8a and 8b show the time it took our incremental ranking algorithm to produce the top-k answers (we exclude cause generation time) for the Homeless scenarios for different error set sizes and instance sizes. Except for the trivial case (singleton error set), the naive algorithm did not finish even computing the top-1 answer in the time limit we had set (10 min). Thus, we do not report any numbers for this approach. Ranking runtime is affected by instance size as query execution and score computation (dealing with larger coverage) is more expensive. Another major cost factor is the input error set size, because the solution space may be exponentially large in this parameter. We also ran the same experiment with the Amalgam scenarios (varying instance size between 200 and 10k) with similarly good results (Figures 8c and 8d). We show the runtime of the naive algorithm below (a cell in red indicates that the algorithm did not finish in 10 min). The second table below shows how many solutions (CCS) have been produced in 10 min. For instances where the algorithm terminates within 10 min this is the total number of CCS that exist. For cases where the algorithm did not terminate, the actual number of CCS can be much larger. This illustrates the complexity of the problem at hand. Thus, it is not surprising that the naive algorithm takes minutes even for a medium sized instance and small amounts of errors. Our approach returns the top-100 ranked causes in seconds (or below) for all settings in both scenarios.

Amalgam - Runtime Naive Method

Instance Size	1 Error	5 Errors	50 Errors	100 Errors
200	0.005	0.104	0.546	1.660
1k	0.021	0.669	16.540	343.678
5k	0.009	86.793	87.252	
10k	0.031	58.498		

Amalgam - # CCS Solutions (within 10 min)

Instance Size	1 Error	5 Errors	50 Errors	100 Errors
200	4	64	448	1,792
1k	4	64	4,864	66,304
5k	4	4,864	4,864	109,461
10k	4	1,792	15,685	32,146

7.3.2 *Impact of Partitioning*

We now compare performance of our incremental ranking algorithm with and without partition ranking. We used a scenario consisting of 10 denormalization primitives (inverse of vertical partitioning). The graph for this scenario has 10 connected components. We have varied the error set size (up to 300 errors) and number of components covered by the error set (higher component numbers should be more beneficial for partition ranking). Figure 8e shows results for a fixed error set size of 20 errors (per component) and varying number of covered components of the scenario graph. Figure 8f shows runtime for producing the top-20 solutions for a fixed number of 5 covered components and varying error set size. Missing bars for ranking without partitioning indicate that the algorithm did not finish within 10 min or ran out of memory. This experiment demonstrates that partitioning can significantly improve performance being able to return result in seconds while the variant without partition runs longer than 10 min.

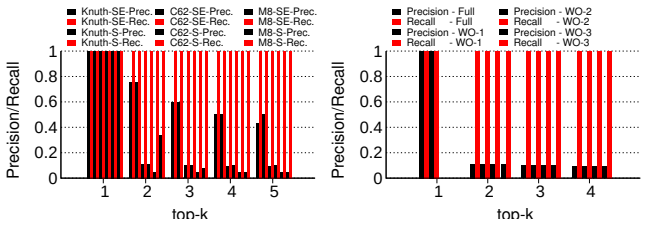
7.4 Ranking Quality

To evaluate the quality of ranking according to our scoring functions, we manually generated errors with known causes (and, hence, also coverage), e.g., by considering a tgd superfluous and computing which tuples have this tgd in their mapping provenance. Thus, correct cause set Λ_G and E_G = COVER(Λ_G) are known and can be compared to the output produced by our ranker. For this experiment we consider the *Amalgam* scenario with 20K tuples. We define the quality of a ranking as the recall and precision of causes in the top-k ranked cause sets, i.e., let Λ_i be the i^{th} ranked cause set then the precision of the top-k ranked cause sets is the number of correct causes (causes in Λ) divided by the total number of causes in the union of the top-k sets. Similarly, recall is the number of correct causes in the top-k cause sets divided by the number of causes in Λ:

$$Prec_k = \frac{\| \bigcup_{i=0}^{k} \Lambda_k \cap \Lambda \|}{\| \bigcup_{i=0}^{k} \Lambda_k \|} \qquad Rec_k = \frac{\| \bigcup_{i=0}^{k} \Lambda_k \cap \Lambda \|}{\| \Lambda \|}$$

7.4.1 *Quality using the full error set*

In this first experiment, we use the total coverage E_G of all correct causes in Λ_G as the input for cause generation and ranking. Thus, this experiment evaluates how well the ranking performs if we have full information about the errors. We considered the following errors: 1) a set of errors containing misspellings of the name Donald Knuth in the target instance's `Author` relation. The cause of these errors are data copy errors; 2) we considered the correspondence relating the title attribute of `MiscPublications` in the source to the attribute title of relation `Article` in the target to be



(a) 100% Errorset **(b)** Errorset of Decreasing Size

Figure 9: Ranking Quality

incorrect; 3) we consider all authors that are not related to any publication in the target to be incorrect. The cause of this error is a mapping that creates these spurious authors. Figure 9a shows the precision and recall achieved by ranking on scoring functions f_{SE} and f_{Size} for increasing sizes of k. Both ranking on side-effect and cause size is very effective in this use case - both returned the correct cause first. Thus, we raised the bar and considered a set of errors which are caused by a set of causes instead of just a single cause: we have combined the Knuth with the correspondence error with the same result.

7.4.2 Quality with partital error set

We now use subsets of the ground truth error set E_G to evaluate how well the ranking performs if only incomplete information is available about the error set. We start with the full set for the correspondence error (2) from above and then randomly remove cells to get subsets of decreasing size. Figure 9b shows how precision and recall are effected if we only give a subset of the errors to Vagabond.

8. CONCLUSIONS

We have introduced an approach for automatically detecting potential causes for target data errors in data exchange. Our approach is unique in that we can explain errors which do not manifest as inconsistencies in the data exchange scenarios and considers errors in the data, the correspondences, and the mapping itself. We presented efficient ranking techniques for causes to present more likely solutions first to the user and avoid exploring the exponentially large search space. While ranking is NP-hard in general, using pruning, partitioning, and additional optimization we can scale to realistically sized datasets, schemas and mappings. In future work, we will study how to automatically suggest repairs for errors detected by Vagabond and how to enable the user to confirm partial solutions as correct and incrementally adapt the solutions and ranking accordingly.

9. REFERENCES

[1] B. Alexe, L. Chiticariu, R. J. Miller, and W.-C. Tan. Muse: Mapping Understanding and Design by Example. In *ICDE*, pages 10–19, 2008.

[2] B. Alexe, L. Chiticariu, and W. Tan. SPIDER: a schema mapPIng DEbuggeR. In *PVLDB*, pages 1179–1182, 2006.

[3] B. Alexe, B. ten Cate, P. Kolaitis, and W. Tan. Designing and refining schema mappings via data examples. In *SIGMOD*, pages 133–144, 2011.

[4] B. Alexe, B. ten Cate, P. Kolaitis, and W. Tan. Eirene: Interactive design and refinement of schema mappings via data examples. *PVLDB*, 4(12):1414–1417, 2011.

[5] P. C. Arocena, M. D'Angelo, B. Glavic, and R. J. Miller. iBench First Cut. Technical report, U. of Toronto, 2013.

[6] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *TODS*, 33(4):31, 2008.

[7] M. Blow, V. R. Borkar, M. J. Carey, C. Hillery, A. Kotopoulis, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, and T. Westmann. Updates in the aqualogic data services platform. In *ICDE*, pages 1431–1442, 2009.

[8] A. Bonifati, E. Chang, A. Lakshmanan, T. Ho, and R. Pottinger. Heptox: marrying xml and heterogeneity in your p2p databases. In *VLDB (demo)*, pages 1267–1270, 2005.

[9] S. Borzsony, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[10] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD*, pages 445–456, 2014.

[11] J. Cheney. Causality and the semantics of provenance. In *DCM*, pages 63–74, 2010.

[12] 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.

[13] L. Chiticariu and W.-C. Tan. Debugging Schema Mappings with Routes. In *VLDB*, pages 79–90, 2006.

[14] J. Chomicki, P. Ciaccia, and N. Meneghetti. Skyline queries, front and back. *SIGMOD Record*, 42(3):6–18, 2013.

[15] X. L. Dong and D. Srivastava. Compact explanation of data fusion decisions. In *WWW*, pages 379–390, 2013.

[16] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[17] R. Fagin, P. Kolaitis, and L. Popa. Data Exchange: Getting to the Core. *TODS*, 30(1):174–210, 2005.

[18] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[19] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, U. of Zurich, 2010.

[20] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the same Data Model through Query Rewriting. In *ICDE*, pages 174–185.

[21] B. Glavic, G. Alonso, R. J. Miller, and L. M. Haas. TRAMP: Understanding the Behavior of Schema Mappings through Provenance. *PVLDB*, 3(1):1314–1325, 2010.

[22] B. Glavic, J. Du, R. J. Miller, G. Alonso, and L. M. Haas. Debugging Data Exchange with Vagabond. *PVDLB (demo)*, 4(12):1383–1386, 2011.

[23] M. Herschel and M. Hernandez. Explaining Missing Answers to SPJUA Queries. *PVLDB*, 3(1):185–196, 2010.

[24] G. Karvounarakis and T. Green. Semiring-annotated data: Queries and provenance. *SIGMOD Rec.*, 41(3):5–14, 2012.

[25] A. Kementsietsidis and M. Wang. Provenance Query Evaluation: What's so Special about it? In *CIKM*, pages 681–690, 2009.

[26] B. Marnette, G. Mecca, P. Papotti, S. Raunich, and

D. Santoro. ++Spicy: an Open-Source Tool for Second-Generation Schema Mapping and Data Exchange. *PVLDB (demo)*, 19:21, 2011.
[27] G. Mecca, P. Papotti, and S. Raunich. Core schema mappings. In *SIGMOD*, pages 655–668, 2009.
[28] A. Meliou, W. Gatterbauer, K. Moore, and D. Suciu. The Complexity of Causality and Responsibility for Query Answers and non-Answers. *PVDLB*, 4(1):34–45, 2010.
[29] A. Meliou and D. Suciu. Tiresias: The database oracle for how-to queries. In *SIGMOD*, pages 337–348, 2012.
[30] R. J. Miller, D. Fisla, M. Huang, D. Kymlicka, F. Ku, and V. Lee. The Amalgam Schema and Data Integration Test Suite, 2001.
[31] R. J. Miller, L. M. Haas, M. A. Hernández, R. Fagin, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exange. *Conceptual Modeling: Foundations and Applications*, page 236, 2009.
[32] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, 2001.
[33] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.
[34] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and Querying Data Transformations. In *ICDE*, pages 81–92, 2005.
[35] Z. Wang. Efficient Scoring and Ranking of Explanation for Data Exchange Errors in Vagabond. Master's thesis, IIT, 2014.

APPENDIX

A. PROOFS

PROPOSITION A.1. *The size of $CCS(E)$ for an error set E can be exponential in the number of elements of E.*

PROOF. Consider a mapping consisting of a single tgd $\sigma : R(a,b) \wedge S(b,c) \rightarrow T(a,c)$ over relations $R(A_1, A_2)$, $S(A_3, A_4)$, and $T(A_5, A_6)$. Let $E = \{T.t_1.A_6, \ldots, T.t_n.A_6\}$ be an error set of size n for some tuples t_1 to t_n in a target instance J. Assume that there exists exactly one grounding for each error in E by constructing the instance so that each tuple from R joins with one tuple from S and vice versa. For each unique grounding for an $e \in E$ there exist at least two explanations: (1) a copy error which explains the error by assuming that $S.t_i.A_4$ is wrong and (2) a source join error that explains the error by assuming the join attribute value $R.t'.A_2$ is incorrect. Thus, we have at least two unique explanations for each error. The number of CES generated by choosing one of the two for each error is already exponential. □

PROPOSITION A.2 (MONOTONICITY OF f_{Size} AND f_{SE}). *Scoring functions f_{SE} and f_{Size} are weakly monotone, but neither monotone nor strongly monotone.*

PROOF. Weak monotonicity follows from the definition of side-effects and semantics of sets in general. A counter example for monotonicity has been given above. □

THEOREM 9 (CES RANKING COMPLEXITY). *Let E be an error set. Generating the top-k ranked CES from enumeration $CCS(E)$ according to scoring functions f_{SE} and f_{Size} is NP-hard in the size of E for any $k \geq 1$.*

PROOF. We prove this theorem through an reduction from the NP-hard *minmal set cover* problem. Let $\mathcal{S} = \{S_1, \ldots, S_n\}$ be a set of sets where each set S in $\mathcal{S}$ is a subset of a universe D so that $\bigcup_{S \in \mathcal{S}} S = D$. A set cover C is a subset of $\mathcal{S}$ so that the union of all sets in C is the universe D. A solution to the minimal set cover optimization problem is a cover with minimal size. We first show a reduction for f_{Size} and then extend this to a reduction for f_{SE}. Given an instance of the minimal set cover problem, i.e., a set $\mathcal{S}$ and universe D, we create a CES ranking problem and show that the top-1 ranked CES for this problem corresponds to a solution for the set cover problem for $\mathcal{S}$. For each element e_i in D we create an error e_i with the a single unique grounding (thus, we can ignore groundings and simply use e_i to denote e_i and its unique grounding). Let E denote the set containing all these errors. For each set $S_i \in \mathcal{S}$ we create an explanation λ_i that explains all errors corresponding to elements in the set, i.e., $\lambda_i \rightsquigarrow e_j$ if $e_j \in S_i$. The generated CES ranking problem is obviously of linear size. An CES for the transformed instance would be a set Λ of explanations so that $\forall e \in E : \exists \lambda \in \Lambda : \lambda \rightsquigarrow e$. That is all errors are explained by at least one explanation in Λ. According to the construction of the CES problem this means that each element in D (error) is contained in at least one set (explanation). Thus, a CES represents a cover for the input set cover problem. Since, f_{Size} ranks on the size of Λ, the top-1 ranked CES is the CES with minimal size (number of explanations). It follows that the subset of $\mathcal{S}$ corresponding to Λ is a solution to the minimal set cover problem for the input instance and, thus, generating the top-1 answer according to f_{Size} is NP-hard. NP-hardness for top-k follows immediately.

For score function f_{SE} we use a similar transformation. Errors correspond to elements in the universe and explanations to sets. In addition we generate a single unique side-effect for each explanation λ. Hence, the number of side-effects is equal to the number of explanations in an explanation set. The remainder of the proof is analog to the case of f_{Size}. □

LEMMA A.1. *Let f be a weakly monotone scoring function and Λ an explanation set explaining some errors in E. The following holds for every CES Λ' that is an extension of Λ.*

$$\text{LOWER}(\Lambda) \leq f(\Lambda') \leq \text{UPPER}(\Lambda)$$

PROOF. We have to show that for any extension Λ' of Λ to a CES the score $f(\Lambda')$ falls between LOWER(Λ) and UPPER(Λ). Let λ_j denote an explanation from Λ' that covers error e_j with $j > i$. The minimal score for the singleton set for each such λ_j is the explanation for error e_j with the minimal score: $\min_{\lambda \in \Lambda_j}(f(\{\lambda\}))$. Applying the inequality of weak monotonicity, the score of any combination of $\Lambda \cup \{\lambda_j\}$ can be bound by $f(\Lambda) \leq f(\Lambda \cup \{\lambda_j\}) \leq f(\Lambda) + f(\{\lambda_j\})$. By symmetry (exchanging Λ and λ_j) we get $max(f(\Lambda), f(\{\lambda_j\})) \leq f(\Lambda \cup \{\lambda_j\}) \leq f(\Lambda) + f(\{\lambda_j\})$. Using induction we get the lower and upper bounds of Lemma A.1. □

THEOREM 10. *If f is a weakly monotone scoring function, then Algorithm 1 returns the top-k ranked explanations for input error set E according to f.*

PROOF. In each iteration of the main loop, one explanation set of size i is replaced with one or more explanation sets of size $i+1$. The only exception is if $i = n$. In this case the explanation set is removed from the queue. Thus, the algorithm eventually terminates for any (finite) input. It remains to show that the returned k explanation sets are the top-k ranked CES according to f. Trivially, each answer returned by the algorithm is a CES, because it covers all errors in E. Note that in the initial state of the priority queue, all CES for $\{e_1\}$ are sorted according to their lower bound LOWER. All potential CES for E are extensions of one (or more) of these singleton explanation sets, because a CES for E covers $e_1 \in E$. Each loop iteration of the algorithm preserves the order (since we are inserting into a priority queue). Now assume at some point in time an explanation set Λ^1 of size n is the first element in the priority queue and, thus, will be returned. We have to show that $f(\Lambda^1) \leq f(\Lambda')$ for all $\Lambda' \in CCS(E)$. We prove this fact by contradiction. Assume there exists such a Λ' with $f(\Lambda') > f(\Lambda^1)$. Since $\|\Lambda'\| = \|\Lambda^1\| = n$ we know that $\text{LOWER}(\Lambda') = f(\Lambda') = \text{UPPER}(\Lambda')$ and $\text{LOWER}(\Lambda^1) = f(\Lambda^1) = \text{UPPER}(\Lambda^1)$ holds. Thus, $\text{LOWER}(\Lambda') \leq \text{LOWER}(\Lambda^1)$. Any extension of an explanation set Λ of size $i < n$ to an explanation set Λ' of size $i+1$ can only increase, but never decrease the lower bound: $\text{LOWER}(\Lambda) \leq \text{LOWER}(\Lambda')$. This fact follows immediately from the definition of the lower bound property. Let $\Lambda[i]$ denote the subset of a CES Λ for E that only contains explanations for errors e_1 to e_i. Consider the loop iteration before Λ^1 was returned. We know that in this iteration $\Lambda^1[n-1]$ was extended to Λ^1 (and potentially to additional CES too). At this point some prefix of $\Lambda'[j]$ of length j has to be on the queue. This follows from the fact that $\Lambda'[1]$ is in the initial queue and if this explanation set was extended, then one of the extensions is $\Lambda'[2]$. By induction there has to exist some $1 \leq j < n$ so the $\Lambda'[j]$ is on the queue. We know that $\text{LOWER}(\Lambda'[j]) \leq \text{LOWER}(\Lambda') = f(\Lambda') < \text{LOWER}(\Lambda^1) = f(\Lambda^1)$. Thus, after the extension of $\Lambda^1[n-1]$ to Λ^1, $\Lambda'[j]$ has to be before Λ^1 in the priority queue which contradicts the fact that Λ^1 is the first element. Using the same argument we can prove that the n^{th} element returned by the algorithm is the n^{th} ranked CES according to f. □

PROPOSITION A.3 (PART. MONOTON. OF f_{SE} / f_{Size}). *Scoring functions f_{SE} and f_{Size} are strongly partition monotone.*

PROOF. Follows from the definition of independent partitioning. □

THEOREM 11 (SG PARTITIONING IS INDEPENDENT). *Let E be an error set for a DES $\mathbb{M}$. A scenario graph partitioning $P_{\mathbb{M}}$ of E is an independent partitioning.*

PROOF. This theorem can be proven by induction over the explanation types. For each combination of two explanation types T_1 and T_2 we have to show that any two explanations of these types for any two groundings for errors from different components of $SG(\mathbb{M})$ are independent (e.g., neither their coverage nor their side-effects do overlap). As an example consider two source copy errors $\lambda_1 = (T_{SC}, O_{I_1})$ and $\lambda_2 = (T_{SC}, O_{I_2})$. We have to prove that $\text{COVER}(\lambda_1) \cap \text{COVER}(\lambda_2) = \varnothing$ and $O_1 \cap O_2 = \varnothing$. First we show that $\text{COVER}(\lambda_1) \cap \text{COVER}(\lambda_2) = \varnothing$. This is proven by showing that the attributes corresponding to every two pairs of errors $R.t.A$ and $R'.t'.A'$ covered by a source copy error λ have to belong to the same component in $SG(\mathbb{M})$. Afterwards, we show that the same applies for the erroneous values in the source which effectively proves $O_1 \cap O_2 = \varnothing$. Consider the definition of a source copy error λ. The coverage contains all target values e' so that there exist a source value e'' in O_I which is in the copy provenance of e'. Assume λ_1 is explaining $R.t.A$. For any source value $e'' = S.B.s$ in O_1, there has to exists a path from $R.A$ to a $S.B$ passing through a mapping variable $\sigma.V$ of some mapping σ. This follows from the definition of copy provenance. Furthermore, for each target value $R'.t'.A'$ covered by λ_1 there has to exist a path from $S.B$ from some source value $e'' \in O_1$ to $R'.A'$ (also follows from the definition of copy provenance). Thus, for each pair of target attributes from the coverage of λ_1 and/or source attributes corresponding to elements from O_I, the attribute and mapping vars belong to the same connected component. If follows that neither the coverage nor the invalidated objects O of two source copy error explanations from different components can possibly overlap. Proving this property for other combinations of explanation types is straight-forward. □

THEOREM 12 (CORRECTNESS OF PARTITIONRANK). *Let P be an independence partitioning of an error set E and f a partition monotone scoring function. Algorithm PartitionRank produces a top-k for E according to f.*

PROOF. We first prove that given two CES Λ and Λ', if Λ dominates Λ' ($\forall i \in \{1,\ldots,n\} : v_\Lambda[i] \leq v_{\Lambda'}[i]$) then $f(\Lambda) \leq f(\Lambda')$. We know that

$$f(\Lambda) = \sum_{i=0}^{n} \Lambda_i[v_\Lambda[i]] \qquad \text{(partition monotonicity)}$$

Furthermore, since the CES for individual partitions are ranked on f we know that $\Lambda_i[j] \leq \Lambda_i[j']$ if $j < j'$. Given that Λ dominates Λ' it follows that $f(\Lambda) \leq f(\Lambda')$. We prove the correctness of the algorithm by induction.

Induction Start: At the beginning $v_1 = [1,\ldots,1]$ is the only element in Q and from the domination property we just have proven follows that v_1 represents the top-1 CES according to f.

Induction Step: Assume we have already produced the top-m solutions. We have to prove that the next iteration of the algorithm generates the $m+1$ ranked solution. At this point in time, the first m elements in Q correspond to the top-m solutions. Q may contain additional elements that have been produced by expanding each of the first m vectors. In the current iteration we insert all expansions of v_m (the m^{th} vector in Q) into Q. We have to show, that (1) all of these expansions will be inserted after v_m and (2) that after the insertion v_{m+1}, the director successor v_m, is the $m+1^{th}$ ranked solution. Since each of the extensions of v_m is dominated by v_m the first claim trivially holds. We prove (2) in two steps. Note that the $m+1^{th}$ solution has to be an direct expansion of one of the top-m solutions. This is easily shown by contradiction. Assume that v_{m+1} is no direct expansion of any of the top-m solutions. Thus, for a given v_i with $i < m$ that can be expanded to v_{m+1} (at least on such v_i has to exist) we can create a sequence of expansions that create v_{m+1} from v_i. WLOG assume the sequence is $v_i, v_i', v_i'', \ldots, v_{m+1}$ where none of the elements in the sequence is one of the top-m vectors. This contradicts the fact that v_{m+1} is the $m+1^{th}$ solution, because v_i' dominates v_{m+1}.

Furthermore, since we have added each expansion of any v_i with $i < m$ during the first $m-1$ iterations and Q is sorted according to f it follows that v_{m+1} has to be the top-$m+1^{th}$ solution. The algorithm will terminate on any finite input, because *ranked* is increased in every loop iteration. □

THEOREM 13 (COMPLEXITY OF PARTITIONRANK). *Given an independent partitioning of an error set E and an (incremental) ranking for each partition according to a strongly partition monotone scoring function f. Algorithm PartitionRank generates a top-k ranking for E from the per-partition rankings in $O(k \cdot \|P\| \cdot log(k \cdot \|P\|))$.*

PROOF. Producing the top-k answers requires k loop iterations. In each iteration we execute at most $\|P\|+1$ lookups and insertions on both Q and *Done*. We implement *Done* using a hash table and Q ($O(1)$ for lookup and insertion) and Q using a tree data structure with $log(\|Q\|)$ for lookups and insertions. Since we add up to $\|P\|$ elements in each iteration the size of Q is bound by $i \cdot \|P\|$ in the i^{th} iteration. Thus, after k iterations $\|Q\|$ is bound by $k \cdot \|P\|$. Thus, the complexity of *PartitionRank* is $O(k \cdot \|P\| \cdot log(k \cdot \|P\|))$. □

THEOREM 14 (BOUND ON PER-PARTITION RANKING). *To produce the top-k CES, Algorithm PartitionRank only accesses per-partition CES that are one of the top-k CES of their partition.*

PROOF. Algorithm *PartitionRank* produces the top-k answers in k iterations. In each iteration i the sorted list Q only contains direct extensions of vectors produced in the previous $i-1$ iterations. The initial queue content is $[1,\ldots,1]$. Thus, by induction after k iterations, for each element $[v_1,\ldots,v_n]$ on the queue we know that $v_i \leq k$ for $i \in \{1,\ldots,n\}$. □