Explaining Query Answers with Explanation-Ready Databases

Sudeepa Roy*
Duke University
sudeepa@cs.duke.edu

Laurel Orr
University of Washington
ljorr1@cs.washington.edu

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

With the increased generation and availability of big data in different domains, there is an imminent requirement for data analysis tools that are able to 'explain' the trends and anomalies obtained from this data to a range of users with different backgrounds. Wu-Madden (PVLDB 2013) and Roy-Suciu (SIGMOD 2014) recently proposed solutions that can explain interesting or unexpected answers to simple aggregate queries in terms of predicates on attributes. In this paper, we propose a generic framework that can support much richer, insightful explanations by preparing the database offline, so that top explanations can be found interactively at query time. The main idea in such *explanation-ready databases* is to pre-compute the effects of potential explanations (called *interventions*), and efficiently re-evaluate the original query taking into account these effects. We formalize this notion and define an *explanation-query* that can evaluate all possible explanations simultaneously without having to run an iterative process, develop algorithms and optimizations, and evaluate our approach with experiments on real data.

1. INTRODUCTION

With the increased generation and availability of large amounts of data in different domains, a range of users, *e.g.*, data analysts, domain scientists, marketing specialists, decision makers in industry, and policy makers in the public sector intend to analyze such data on a regular basis. They explore these datasets using modern tools and interfaces for data visualization, then try to understand the trends and anomalies they observed in their exploration so that appropriate actions can be taken. However, there are currently no tools available that can automatically 'explain' trends and anomalies in data. As more users interact with more datasets, the need for such tools will only increase.

A couple of recent research projects by Wu and Madden [34] and Roy and Suciu [32] have proposed techniques for explaining interesting or unexpected answers (*e.g.*, outliers) to a query Q on a database D. For instance: *why is the average temperature reported by a number of sensors between 12PM and 1PM unexpectedly high [34]?*, or, *why does the number of SIGMOD publications from industry have a peak during the early years of the 21st century [32]?* Both [34] and [32] consider an explanation to be a *conjunctive predicate* on the input attributes, *e.g.*, [`sensorid` = 18], or [`author.institution` = 'Oxford' `and paper.year` = '2002']. An explanation predicate is considered good for a particular trend or outlier if, by removing from the database all tuples that 'depend on' the predicate, the trend changes or the outlier is eliminated. In the sensor data example, the explanation [`sensorid` = 18] means that, if we removed all readings from this sensor, then the average temperature between 12PM and 1PM is no longer high. We give a new example in this paper from the NSF awards dataset[1].

EXAMPLE 1.1. *The three main tables in the NSF awards dataset are as follows (the keys are underlined):*

Award(aid, amount, title, year, startdate, enddate, dir, div)*;*
Institution(aid, instName, address)*;*
Investigator(aid, PIName, emailID)

Here dir *denotes the* directorate*, i.e., the area like Computer Science (CS), and* div *is the* division *within an area, e.g.,* CCF, CNS, IIS, ACI *for CS. Now consider the query below:*

```
SELECT TOP 5 B.instName, SUM(A.amount) AS totalAward
FROM Award A, Institution B
WHERE A.aid = B.aid AND dir = 'CS' AND year >= 1990
GROUP BY B.instName
ORDER BY totalAward DESC
```

The above query seeks the top-5 institutions with the highest total award amount in CS from 1990, and produces the following answer:

instName	totalAward
University of Illinois at Urbana-Champaign	1,169,673,252
University of California-San Diego	723,335,212
Carnegie-Mellon University	472,915,775
University of Texas at Austin	319,437,217
Massachusetts Institute of Technology	292,662,491

*This work was done while the author was at the University of Washington.

[1]This publicly available dataset [1] contains awards from 1960 to 2014 in XML that we converted into relations. We omit some tables and attributes, and use abbreviations for simplicity, *e.g.*, the directorate of CS will appear as 'Directorate for Computer & Information Science & Engineering'.

Proceedings of the VLDB Endowment, Vol. 9, No. 4

If someone is interested in analyzing the NSF funding awarded to different schools, looking at the above answers, she might try to understand ***'why there is a huge difference in award amounts between*** `UIUC` ***and*** `CMU`*', especially given that* `CMU` *holds rank-1 as a CS graduate school and* `UIUC` *holds rank-5 (according to US News [2]).*

Indeed, the algorithms in both [34] and [32] can be used to explain the difference in award amounts between `UIUC` and `CMU`; since [34] operates on a single relation, it will first materialize the join between the *Award* and *Institution* tables. Both the approaches return predicates on the attributes of these two tables as explanations, *e.g.*, $[div = 'ACI']$ (Advanced Cyberinfrastructure). Intuitively, if the awards from this division are removed (which amounts to about \$893M for `UIUC` and only \$26M for `CMU`), then the difference between the award amounts for `UIUC` and `CMU` will drastically change.

Following the *causality* literature [29], the act of removing tuples from a database D and then studying its effect on the answer to a query Q (not physically, only to evaluate an explanation) is called an *intervention*. An explanation e has a high score if $Q[D - \Delta D_e]$ differs significantly from $Q[D]$, where $\Delta D_e \subseteq D$ refers to the tuples in the intervention of e. An explanation system considers many candidate explanations e, computes some score based on the amount of change in the query after the intervention, and returns to the user a ranked list of explanations.

However, for performance reasons, current explanation systems severely limit the types of explanations that they search, and fail to find deep and insightful explanations. For instance, an interesting explanation is the PI ***'Robert Pennington'***, who received more than \$580M for `UIUC` in CS. His absence from the database will hugely affect the difference between the award amounts of `UIUC` and `CMU` (assuming an award depends on all PIs). Although this explanation is expressible as a predicate [*PIName* = 'Robert Pennington'], both [34, 32] fail to return this explanation simply because the PI information belongs to the third table *Investigator*, which does not appear in the query in Example 1.1.

In particular, Wu and Madden [34] limit explanations to predicates on a single table (base relation or a materialized result), thereby possibly missing explanations that depend on mutual dependency of multiple tables in a database. Even if all three tables are joined and materialized a priori, the intervention of predicates like [*PIName* = 'Robert Pennington'] (removal of all awards for this PI) cannot be computed by removing tuples satisfying the predicate, since there will be other rows for awards with multiple PIs that do not satisfy this predicate. On the other hand, Roy and Suciu [32] can support predicates spanning multiple tables and dependency between tables through foreign keys and reverse foreign keys as causal dependencies — if a tuple with a primary key is deleted, the corresponding tuple with the foreign key is deleted (and vice versa for reverse foreign keys). But computing the effect of an intervention, in the worst case, requires running a non-trivial recursive query. Therefore, they restrict the predicates to conjunctive equality predicates on the tables used in the query so that the efficient OLAP data cube operation in SQL can be used to increase performance. Moreover, both [34] and [32] restrict the class of input queries to single-block aggregate queries (of the form *select-from-where-group by* as in Example 1.1) without any nested sub-queries, which cannot express some important statistical methods frequently used in data analysis (*e.g.*, linear regression or correlation coefficient on intermediate aggregates, an example is given in Section 5).

In this paper we propose a new approach for finding deep, semantically meaningful explanations with interactive performance, which we call **Explanation-Ready Databases (ERD)**. Our proposal is based on two ideas. First, we *precompute* the interventions associated with a large number of potential explanations and store them in the database. Note that the intervention is defined independently of a query, and therefore it can be precomputed. We use arbitrary causal dependencies induced by the semantic of the application to associate interventions to candidate explanations (ref. Section 3.1). Second, at query time, we compute the scores of *all* candidate explanations simultaneously, by a single SQL query called the *explanation-query*, which is evaluated on both the original data and the interventions associated to all explanations. We present a suite of incremental query computation techniques that allow us to compute the explanation query at an interactive speed.

EXAMPLE 1.2. *For the explanation* e : [*PIName* = 'Robert Pennington'], $\Delta D_e = (\Delta_{Award,e}, \Delta_{Institution,e}, \Delta_{Investigator,e})$ *consists of interventions on all three tables. PIs do not affect the presence of their institutions in the database, therefore* $\Delta_{Institution,e} = \emptyset$. $\Delta_{Investigator,e}$ *contains all tuples from* Investigator *such that PIName =* 'Robert Pennington' *and* $\Delta_{Award,e}$ *contains all awards with* 'Robert Pennington' *as a PI.* Δ_{Award} *can be populated with interventions of all such explanations* e *(indexed by* e*) by a nested SQL query that joins the* Award *and* Investigator *tables.*

A major advantage of an explanation-ready database is that it can consider much richer explanations of different forms, because the high cost of computing their intervention is paid offline (*e.g.*, by running multi-block SQL queries with nested aggregates and joins). Examples include:

- **Explanations involving combination of attributes and tables:** *e.g.*, awards with duration $\geq x$ years or awards with the number of PIs $\geq y$. The first one requires taking the difference between *startdate* and *enddate* whereas the latter requires a join with the *Investigator* table that does not participate in the original query.
- **Explanations with aggregates:** *e.g.*, PIs with $\geq X$ awards, $\geq Y$ co-PIs, or $\geq Z$ total award amount. This requires joining the *Investigator* table with itself or with the *Award* table and computation of aggregate values.
- **Explanations having top-k form:** If one or more from the top-k PI-s (according to total award amount, average award amount, etc.) across all institutions belong to `UIUC`, they can explain the high award amounts of `UIUC`.

To improve the performance of the explanation query we adapt techniques from *Incremental View Maintenance (IVM)*. In Section 2, we discuss the similarities and differences between IVM and an ERD. Specifically, our task is to compute $Q[D - \Delta D_e]$ incrementally, from $Q[D]$ for all e. In IVM, the goal is to re-compute *a given query* when a single tuple is deleted from the database, or a subset of tuples for batch deletion. In our case, (i) the query is only known at runtime, and (ii) not only the intervention ΔD_e contains many tuples from many relations, there can be thousands of such ΔD_e-s; in fact, the union of all ΔD_e-s for all candidate explanations e can be much larger than the database D,

which we account for in our incremental techniques. We also observed that iteratively running IVM for all explanations e (even using a state-of-the-art IVM tool like DBToaster [9]) takes time that increases rapidly with the number of explanations, while our incremental approach of running the single explanation-query reduces the running time by an order of magnitude. For the question in Example 1.1, the explanation-query considered more than 188k explanations and returned the top explanations in < 2 seconds. An ERD incurs an additional space cost in order to store the interventions for a set of explanations. Although the actual cost depends on the number of explanations considered and size of their interventions depending on the application, our experiments suggest that this cost is manageable and is a small price to pay for deep understanding of data. Moreover, we only need to store interventions of complex explanations for single-block queries as our technique can be used in conjunction with the techniques in [34, 32] that return simple explanation predicates on the fly. Here is a summary of our contributions in this paper:

- We propose the notion of explanation-ready databases (ERD), which store the intervention associated with all candidate explanations (Section 3).
- We describe a suite of techniques inspired by IVM for computing the explanation-query incrementally (Section 4).
- We experimentally evaluate our approach using real datasets (Section 5).

2. RELATED WORK

Explanations in databases. Several research projects in databases and data mining aimed to provide explanations in interesting applications, *e.g.*, mapreduce [21], user ratings on sites like Yelp and IMDB [15], access log and security permissions [17, 11], etc.; a survey can be found in a recent tutorial[27]. For aggregate database queries, Wu-Madden [34] and Roy-Suciu [32] proposed frameworks for finding predicates as explanations as discussed in the introduction. In contrast, our goal in this paper is to support broader classes of queries and explanations. The annotated relations described in this paper are similar to the ones in provenance semirings for aggregates studied by Amsterdamer et al.[10]. They describe how the provenance of an aggregate query can be formally recorded, whereas our focus is on recomputing the answer $Q[D]$ of a query for all possible explanations. Related topics for non-aggregate queries that target to understand the query and the data are *causality* [25, 26] (rank individual input tuples according to their *responsibility* toward the existence of given output tuples), *deletion propagation* (delete given output tuples with minimum effect on the source or the view, *e.g.*, [22]), *query by output* [33] (given a query and instance, find an equivalent query for the given instance), *data extraction by examples* [24], *synthesizing view definitions* [16], etc.

Incremental View Maintenance (IVM). IVM, and view management in databases in general, have been extensively studied in the literature (see, *e.g.*, the surveys in [13, 9, 20]). The goal of IVM is to compute the new value of a query from $Q[D]$ for insertion/update $Q(D \cup \Delta D)$ or deletion $Q[D - \Delta D]$ avoiding re-evaluation of the query Q. IVM has been studied for set and bag semantics (*e.g.*,[12, 18]) primarily for single-block queries with or without aggregates, selective materialization of views (*e.g.*, [31]), lazy vs. eager evaluation (*e.g.*, [14]), utilizing primary key constraints in IVM (*e.g.*, [20]), concurrent executions of read-only queries and maintenance transactions on materialized views [30], and for distributed programs in a network [28]. The standard approach for IVM is *delta processing*, where the updates are maintained and propagated to the view when needed [18, 19, 12]. The recent DBToaster project [9, 23, 3] supports complex multi-block queries, and maintains multiple levels of delta relations (higher order deltas) to allow for incremental changes to all the levels of delta relations for each single tuple update in order to be able to update a view for large rapidly evolving datasets. DBToaster compiles the SQL code for a given query into C++ or Scala code, which can be embedded in an application to monitor the query answer when a tuple is inserted and deleted from the database. We have compared our algorithms with DBToaster experimentally in Section 5.

Comparisons of ERD with IVM

At a high level, the technical goals of ERDs and IVM are similar: reduce the complexity of re-computing the same query on slightly different databases. An ERD aims to evaluate $Q[D - \Delta D_e]$ *for all possible* $E = e$, where each $\Delta D_e \subseteq D$ is a set of tuples, which intuitively generalizes the objective of IVM. However, there are several differences between ERDs and IVM:

1. **Deletions in an ERD are hypothetical.** We intend to evaluate the new value of the query *assuming* the tuples in the intervention ΔD_e have been deleted from D. At the end, the database D should be left unchanged in contrast to IVM where the updates affect the database.
2. **IVM focuses on fixed query with dynamic updates, whereas ERD focuses on fixed updates and dynamic queries.** *Given an SQL query*, DBToaster compiles it into a C++ or a Scala program, which provides methods to update the value of the query for each database update. For a new query, the program has to be regenerated again. In contrast, in an ERD, the possible changes ΔD_e-s to the database are pre-determined and fixed, but the query is not known upfront. It will only be known in the runtime, and is likely to change more frequently as the user tries to understand the answers.
3. **There is no need for low-level optimizations in an ERD.** Our algorithms work on standard DBMS as the updates are hypothetical (unlike DBToaster). Therefore, we are able to use the in-built optimizations provided by a DBMS, and need not consider low-level implementation details (*e.g.*, new query languages/compilers, selective materialization of intermediate views, caching, and synchronization [9]).
4. **Complexity.** By aggressively pre-processing the query, and maintaining a larger number of intermediate views, IVM in DBToaster has been developed to an extreme such that every update to a materialized view can be computed in NC^0 (Koch [23]). In the explanation framework, the query is only known at runtime, and as a consequence we cannot perform a similar aggressive preprocessing. Instead, we reduce the complexity from re-computing the same query once for each intervention, to computing a single query over a larger database. In general, our algorithms have polynomial data complexity.

The main focus of an ERD is to compute $Q[D - \Delta D_e]$ for all $E = e$ without having to run an iterative 'for loop' on such e-s, which is orthogonal to the goal of the IVM approaches. Our experiments in Section 5 show that iteratively running IVM (*e.g.*, DBToaster) for all $E = e$ does not give good performance. Nevertheless, due to the simililarity with the high level goal of IVM, our techniques are motivated by the IVM literature. For instance, the update rules in Section 4.3 can be considered as extensions to the update rules in IVM [12, 18], although these rules take into account multiple updates to the views *for all* explanations $E = e$ simultaneously, and therefore incur additional complexity (*e.g.*, the union of the ΔD_e relations can be much larger than the original database D in an ERD, which is never the case for IVM).

3. EXPLANATION-READY DATABASES

We describe here the framework for an explanation-ready database (ERD). The offline part is discussed in Section 3.1, and the interactive part in Section 3.2.

3.1 Preparing the Database

An *explanation-ready database* $ERD(D, \mathcal{P}, \mathcal{T}, \mathcal{E})$ (ERD in short) has four components:

- A **standard relational database** D with k relations $D = (S_1, \cdots, S_k)$ for some constant $k \geq 1$. The relation S_i has attributes $\bar{\mathtt{A}}_i$ for[2] $i \in [1, k]$.
- A set of **explanation types** $\mathcal{T}$; each type has an identifier, a short English description, and k SQL queries (one for each relation S_i, discussed below).
- A class of **potential explanations** (henceforth, *explanations*) $\mathcal{E}$, where each explanation $(e, \Delta D_e, \mathtt{type}_e) \in \mathcal{E}$ has three components:
  1. **ID:** A unique integer identifier e.
  2. **Intervention**: A subset of tuples $\Delta D_e = (\Delta_{1,e}, \cdots, \Delta_{k,e})$; each $\Delta_{i,e} \subseteq S_i$, $i \in [1, k]$. By *intervening* the explanation e, we mean removing all tuples ΔD_e from the database D.
  3. **Type:** A $\mathtt{type}_e \in \mathcal{T}$.
- A set of **causal dependencies** $\mathcal{P}$ among tuples using datalog rules (described below).

The ERD is prepared offline, independently of any query, and stored in the databases as follows. There is a separate table storing the types in $\mathcal{T}$. All explanations are stored together in k tables, denoted $\Delta D = (\Delta_1, \cdots, \Delta_k)$, where Δ_i has the same attributes as the relation S_i plus an extra attribute E, storing the explanation identifier e; thus, for each identifier value e, one can recover the intervention associated to e, $\Delta_{i,e}$, through a selection followed by a projection: $\Delta_{i,e} = \pi_{\bar{A}_i} \sigma_{E=e} \Delta_i$. We also create a primary index (non-unique) on the attribute E. Finally, we store separately the many-one relationship associating each explanation identifier e with its type. Here is a toy example:

EXAMPLE 3.1. *Consider a database instance D in Figure 1 with relations S_1, S_2, $\bar{\mathtt{A}}_1 = \{A, B\}$, $\bar{\mathtt{A}}_2 = \{B, C\}$. Consider three explanations with IDs $E = 1, 2, 3$, where ΔD_1 $= \{S_1(a_1, b_2),\ S_2(b_1, 9),\ S_2(b_3, 8)\}$, $\Delta D_2 = \{S_1(a_1, b_1),\ S_1(a_2, b_1),\ S_2(b_1, 9)\}$, and $\Delta D_3 = \{S_2(b_2, 5),\ S_2(b_3, 8)\}$. $\Delta D = (\Delta_1, \Delta_2)$ is shown in the figure.*

[2] We use an overline to denote a *vector* of attributes, annotations, or values; For simplicity, we will interpret the vectors as sets (with components of the vectors as elements of the sets) and use the standard notations $\in, \subseteq, \cup, \cap, \setminus$ etc. For two integers a, b where $a \leq b$, $[a, b]$ denotes $a, a+1, \cdots, b-1, b$.

S_1

A	B
a_1	b_1
a_1	b_2
a_2	b_1

S_2

B	C
b_1	9
b_2	10
b_2	5
b_3	8

Δ_1

E	A	B
1	a_1	b_2
2	a_1	b_1
2	a_2	b_1

Δ_2

E	B	C
1	b_1	9
1	b_3	8
2	b_1	9
3	b_2	5
3	b_3	8

Figure 1: A toy instance of a database D and ΔD.

We now describe how the interventions ΔD_e are computed (offline). Recall that each type $\mathtt{type}_e$ is associated with k SQL queries, one for each relation S_i. Query i returns a set of tuples (e, t), where $t \in S_i$ and e is some value, interpreted as the identifier of an explanation. Taking the union of all types in $\mathcal{T}$, we have some initial tuples in relations Δ_i-s.

Next, the system runs the datalog rules in $\mathcal{P}$, adding to this intervention all tuples reachable by a causal dependency:

$$\Delta_i(e, \bar{x}) :- Y_1(\bar{x_1}), Y_2(\bar{x_2}), \cdots, Y_p(\bar{x_p}) \quad (1)$$

where $i \in [1, k]$, $Y_j \in \{S_1, \cdots, S_k\} \cup \{\Delta_1, \cdots, \Delta_k\}$, for each $j \in [1, p]$, and as standard, each variable in the head of the rule appears in the body. These rules define when a tuple will indirectly cease to exist in the database, due to removal of other tuples.

Thus, the role of the types is two-fold, to provide a brief English description of an explanation e of that type, and to define how to compute the interventions ΔD_e for explanation e of that type. The ERD is managed by a domain expert, knowledgeable about the domain of the database, who defines the types of explanations, their associated queries, and the causal dependencies: this is done offline, independently of what queries will be explained. Notice that one advantage of an ERD is that it allows us to consider *heterogeneous explanations* of different *types* in contrast to the previous work [32, 34]. We illustrate the concepts below:

EXAMPLE 3.2. *For the NSF awards example, the database D contains three relations* `Award, Investigator, Institution` *as shown in Example 1.1. The dependencies in this database are given by the two rules that hold for all e as shown below (if an investigator is deleted, delete all of his/her awards, and if an institution is deleted, delete all awards received by this institution):*

$$\Delta_{\mathtt{A}}(\mathtt{e}, \mathtt{aid}, \mathtt{x}_1, \cdots, \mathtt{x}_p) :- \Delta_{\mathtt{Inv}}(\mathtt{e}, \mathtt{aid}, \mathtt{y}_1, \cdots, \mathtt{y}_\ell), \mathtt{A}(\mathtt{aid}, \mathtt{x}_1, \cdots, \mathtt{x}_p)$$
$$\Delta_{\mathtt{A}}(\mathtt{e}, \mathtt{aid}, \mathtt{x}_1, \cdots, \mathtt{x}_p) :- \Delta_{\mathtt{Ins}}(\mathtt{e}, \mathtt{aid}, \mathtt{z}_1, \cdots, \mathtt{z}_m), \mathtt{A}(\mathtt{aid}, \mathtt{x}_1, \cdots, \mathtt{x}_p)$$

Here x, y, z variables correspond to the attributes of `A = Award, Inv = Investigator` *and* `Ins = Institution` *respectively.*

We considered eight types in $\mathcal{T}$ as shown in Table 1; these explanation provide individual or aggregate information (top-k, ranges) about the three entities in the domain (award, investigator, institution); several other explanation types are possible. The numbers of individual explanations e belonging to each of these types are shown in the third column in the table. For each type, we run queries (1) to find out the individual explanations in that type, (2) to compute the initial set of tuples in the interventions ΔD_e depending on the type, (3) to complete the interventions ΔD_e using the datalog rules in $\mathcal{P}$.

Type	Description	#Expl. e	Avg. $\lvert\Delta D_e\rvert$	$\sum_e \lvert\Delta D_e\rvert$
1	Individual PI from the `Investigator` table	170,619	3	620,656
2	Top-K PIs with highest average award amounts $K = 10, 20, \cdots, 100$	10	148	1,481
3	Top-K PIs with highest total award amounts $K = 10, 20, \cdots, 100$	10	145	1,453
4	Awards with duration (in years) $[0,1); [1,2); [2,3); \cdots; [9,10); [10,-)$	11	36,391	400,303
5	Awards with no. of PIs $1, 2, \cdots, 5, \geq 6$	6	7,830	46,981
6	PIs with no. of awards $[1,2]; [3,4] \cdots, [9,10]; [11,-)$	6	21,858	131,152
7	PIs with total award amounts (in million dollars) $[0,1); [1,5); [5,10); [10,50); [50,100); [100,-)$	6	21,149	126,896
8	Individual institutions from the `Institution` table	17,696	23	418,962

Table 1: Details of eight explanation types and their ΔD_e relations.

For example, the first type contains all the names of all the PIs from the `Investigator` *table (across all schools, not only from* `UIUC` *and* `CMU`*), which can be easily selected and stored with unique IDs into a table* `ExplType_1`*. The English description for this type of explanations is of the form* "The PI `<PIName>`"*. For any PI, say* 'Robert Pennington'*, suppose the unique ID is* 10*. Here* $\texttt{type}_{10} = 1$ *and its description is* "The PI 'Robert Pennington'"*;* $\Delta_{\texttt{Investigator}}$ *includes a single tuple for this PI (indexed with* $E = 10$*);* $\Delta_{\texttt{Institution}}$ *is unaffected; and* $\Delta_{\texttt{Award}}$ *includes all the award tuples for which he is a PI/co-PI (by the first rule in Example 3.2). For all explanations* e *of this type,* $\Delta_{\texttt{Award}}$ *(initialized as an empty table) can be augmented by a single SQL query that combines* `Award, Investigator`*, and* `ExplType_1` *tables. In general, the description of an explanation* e *can be obtained from the description of its type* $\texttt{type}_e = \ell$*, from the information stored in the* $\texttt{ExplType}_\ell$ *tables. Some of the other explanation types* ℓ *are more complex, and will require nested subqueries with aggregates to compute the* $\texttt{ExplType\_}\ell$ *tables and the initial tuples in the interventions.*

3.2 Explaining a Query

Once the explanation-ready database $ERD(D, \mathcal{P}, \mathcal{T}, \mathcal{E})$ is computed and stored in the relations $(\Delta_1, \cdots, \Delta_k)$, the ERD is ready for explaining user questions.

User, Query, User Question:

A *user* issues a *query* Q to the database D and gets a set of tuples $Q[D]$ as the answer (see Example 1.1; we define in Section 4.1 the class of SQL queries supported. If the user finds the value of some of the tuples in $Q[D]$ high or low, she can ask a *user question* (or simply a *question*), *e.g.*, '*why is the total award amounts of* `UIUC` *high*'. Formally, such questions are of the form *'why is* $Q_1[D]$ *high' (or low)'*. Here Q_1 is a simpler aggregate query with a single numeric output (no group by) and an additional selection condition, *e.g.*,

```
SELECT SUM(A.amount) AS totalAward
FROM Award A, Institution B
WHERE A.aid = B.aid AND dir = 'CS' AND year >= 1990
          AND B.instName = 'UIUC'
```

This simpler query Q_1 can be easily generated from the original query Q (see Example 1.1). A good *explanation* e will explain such *'why is* $Q_1[D]$ *high'* (respectively, *low*)' questions by decreasing (respectively, increasing) the value of $Q_1[D - \Delta D_e]$ by its intervention ΔD_e. The user question can also involve multiple tuples in $Q[D]$, *e.g.*, *'why is* $Q_1[D] - Q_2[D]$ *high'*, like in Example 1.1 where the award amounts of `UIUC` and `CMU` are compared. In this case, a good explanation e will have a low value of $Q_1[D - \Delta D_e] - Q_2[D - \Delta D_e]$.

The same ERD for multiple queries and questions:

The user can ask multiple questions for the same query or can ask different queries (and questions thereafter) to the ERD. For instance, the same set of potential explanations outlined in Table 1 can be used to explain a different question on the query answers in Example 1.1:

"Why does `UCSD` *have larger amount of awards than* `MIT`*?"*

On the other hand, the same set of potential explanations can be used for a different database query altogether:

Example 3.3. *We executed an SQL query on the* `Award` *table to compute the the total award amounts and average amount per award (in USD) from the year 1990 in different divisions of the directorate of CS. The answers are:*

Division	*Total amount*	*Average amount*
IIS	*2893.3M*	*365K*
CCF	*2833.9M*	*305K*
CNS	*3637.8M*	*405K*
ACI	*3028.4M*	*1407K*
OTHER	*352.1M*	*332K*

Given that the main four divisions (`IIS, CCF, CNS, ACI`*) have comparable amounts of total awards, the user can ask*

"Why is the average award amount of ACI high?"

After exploring the query/questions in Example 1.1, the ERD can use the same potential explanations in Table 1 to answer questions on this query. Indeed, neither all the explanation types nor all the explanations in a type will be relevant to a query/question combination. For instance, the explanation type#8 (different institutions) is not relevant for Example 1.1 where the institution is a fixed parameter in the question. Similarly, explanations who are PIs from different institutions other than `UIUC` or `CMU` in the explanation type#1 (different PIs) are not relevant.

4. BUILDING EXPLANATION-QUERY

In this section, first we formalize the concept of subtracting a relation from another by defining annotated relations and an algebra (Section 4.1), then define the intervention mapping Δ_Q and its incremental construction along a query plan (Sections 4.2 and 4.3), and then describe the construction of the final explanation-query (Section 4.4). We will use Q to denote the query as well as its output $Q[D]$ when the underlying database D is clear from the context.

4.1 Annotated Relations and Algebra

In order to formalize and compute the difference in the answers $\Delta_Q[D, \Delta D]$, we view the base relations as well as all intermediate and final relations as *annotated relations* that consider aggregate and non-aggregate attributes separately. Let $\mathbb{D}$ be the domain of all attributes in the database D; we assume that the set of reals $\mathbb{R} \subseteq \mathbb{D}$.

DEFINITION 4.1. *An* annotated relation S *of* type $[\bar{\mathtt{A}}; \bar{\mathtt{K}}]$ *is a function* $S : \mathbb{D}^m \to \mathbb{R}^\ell$, *where* $|\bar{\mathtt{A}}| = m$ *and* $|\bar{\mathtt{K}}| = \ell$. *Here* $\bar{\mathtt{A}}$ *and* $\bar{\mathtt{K}}$ *are called the* standard attributes *and the* annotation attributes *of* S *respectively. Further, the* support *of* S, *i.e., the number of tuples in* $\{\bar{t} \in \mathbb{D}^m : S(\bar{t}) \neq \bar{0}^\ell\}$, *is finite.*

Intuitively, $\bar{\mathtt{A}}$ corresponds to the non-aggregate attributes, and the annotation $\bar{\mathtt{K}}$ corresponds to one or more aggregates when the relation has been generated by an aggregate query). Then $S(\bar{t})$ denotes the values of these aggregates for a tuple $\bar{t}$ comprising only standard attributes. Since S is a function, $\bar{\mathtt{A}}$ can be considered as the *key* of S; so, we follow the set semantics and do not allow duplicates in any intermediate relation[3]. The base relations S_i, $i \in [1, k]$, are annotated relations of type $[\bar{\mathtt{A}}_i; \mathtt{K}]$, where for all tuples $\bar{t} \in S_i$, $S_i(\bar{t}) = 1$. The annotation $\mathtt{K}$ simply corresponds to the finite support of the relations and is not physically stored; this also holds for any non-aggregate query. Here $\mathtt{K}$ is called the *trivial annotation* and the 1/0 annotations are treated as Boolean true/false. Otherwise, when $\bar{\mathtt{K}}$ denotes non-trivial annotations (as the result of an aggregate query), we store S in a standard DBMS with attributes $\bar{\mathtt{A}} \cup \bar{\mathtt{K}}$.

Algebra on Annotated Relations

Class of SQL queries: We support a subclass of SQL queries Q that extends the class of single-block "`select-from-where-group by`" aggregate queries considered in previous work [34, 32]. In particular, we allow (i) union operations in non-aggregate queries, and (ii) multiple levels of aggregates, but do not allow selection σ, join $\bowtie$, and union $\cup$ operations on an aggregate sub-query. This fragment of SQL queries can be expressed by a *query plan tree* where all $\sigma, \bowtie, \cup$ operators appear below all aggregate (γ) operators in the plan[4]. Examples include the single-block queries in Examples 1.1 and 3.3, and the nested query in Section 5.2. We do not support queries with bag semantics or non-monotone queries. The grammar for the intended aggregate query class Q_a can be defined as follows (Q_a, Q_{na} respectively denote aggregate and non-aggregate queries):

$$Q_{na} = S_1 \mid \cdots \mid S_k \mid \sigma_c(Q_{na}) \mid Q_{na} \bowtie_c Q_{na} \mid Q_{na} \cup Q_{na} \quad (2)$$

$$Q_a = \gamma_{\bar{\mathtt{A}}, aggr_1(g_1(\bar{B}_1)), \cdots, aggr_s(g_s(\bar{B}_s))}(Q_{na} \mid Q_a) \quad (3)$$

However, the queries Q_{na}, Q_a in the above grammar correspond to annotated relations both as the inputs and outputs. In particular, $S_1, \cdots, S_k$ are the *annotated* base relations in D. For this class of queries, the inputs to the non-aggregate operators are annotated relation(s) with trivial annotation $\mathtt{K}$. Therefore, the predicate c for σ and $\bowtie$ is a predicate on the standard attributes. Due to space constraints, we only give the semantic of the aggregate operator γ (the rest can be found in the full version [4]).

Semantic of the γ operator: *Let* $Q = \gamma_{\bar{\mathtt{A}}, aggr_1(g_1(\bar{B}_1)), \cdots, aggr_s(g_s(\bar{B}_s))} Q'$, *where* Q' *is of type* $[\bar{\mathtt{A}}'; \bar{\mathtt{K}}']$, $|\bar{\mathtt{A}}'| = m$, $|\bar{\mathtt{K}}'| = \ell$, *and each* $\bar{B}_i \subseteq \bar{\mathtt{A}}' \cup \bar{\mathtt{K}}'$. *Then* Q *is of type* $[\bar{\mathtt{A}}; \bar{\mathtt{K}}]$, *where* $|\bar{\mathtt{K}}| = s$ *and* $\bar{\mathtt{K}}$ *stores the aggregates:* $aggr_1(g_1(\bar{B}_1)), \cdots, aggr_s(g_s(\bar{B}_s))$. $\forall \bar{t} \in \mathbb{D}^m$, *the* j*-th component of* $Q(\bar{t})$, $j \in [1, s]$, *is:*

$$aggr_j \quad {}_{\bar{t}' \in \mathbb{D}^m : \bar{t}'.\bar{\mathtt{A}}' = \bar{t}} \quad \left(g_j(\bar{t}'.\bar{B}_j) \times I[Q'(\bar{t}') \neq \bar{0}^\ell] \right)$$

where $I[Q'(\bar{t}') \neq \bar{0}^\ell]$ *is the indicator function denoting whether the tuple* $\bar{t}'$ *has a non-zero annotation and therefore can possibly contribute to the aggregate. Here,* $aggr_i \in \{sum, count, avg, min, max, count\ distinct\}$ *and* g_i *is a scalar function which can be a constant, an attribute, or a numeric function involving* $+, -, \times, min, max$, *or any other function supported by SQL.*

EXAMPLE 4.2. *(1) In* $Q = \gamma_{\bar{\mathtt{A}}, count(*)} = \gamma_{\bar{\mathtt{A}}, sum(1)}$, *the aggregate function is* $aggr = sum$; *the scalar function is* $g(x) = 1$. *(2) In* $Q = \gamma_{\bar{\mathtt{A}}, sum(B_1 \times B_2)}$, $aggr = sum$ *and* $g(x, y) = x \times y$, *which operates on the attributes* $\bar{B} = \langle \bar{B}_1, \bar{B}_2 \rangle$. *(3) In* $Q = \gamma_{\bar{\mathtt{A}}, sum(\min(B_1, B_2))}$, $aggr = sum$, *and* $g(x, y) = \min(x, y)$. *(4) In* $Q = \gamma_{\bar{\mathtt{A}}, \min(\min(B_1, B_2))}$, $aggr = min$, $g(x, y) = \min(x, y)$.

The semantics of the aggregate function and the scalar function are different even if they use the same function like *min*; the scalar function applies to different attributes of the same tuple, whereas the aggregate function applies to all the tuples in a group with the same value of the attributes $\bar{\mathtt{A}}$.

From this point on, we assume that the aggregate function is *sum*; the other operators *count*, *avg*, min / max and *count distinct* can be simulated using *sum* (see [4]). However, if min / max /*count distinct*/*avg* operators appear in an intermediate step in the query plan (which is less common in practice), $Q[D - \Delta D]$ needs to be computed from $Q[D]$ and $\Delta_Q[D, \Delta D]$ before proceeding further; see Section 6. Since annotated relations require set semantic, we do not include the projection operator π in the grammar. A projection on to a set (*e.g.*, select distinct A, B, C from ...) can be captured using the aggregate operation $Q = \gamma_{A,B,C} Q'$, where no aggregate value is computed for each group. The projection with duplicates, which is allowed in standard database systems, is not allowed in our framework.

4.2 Intervention Mapping

In this section we formalize the concepts described in Example 3.1 and define the desired Δ_Q query for a given query Q. Similar to the inputs and outputs of query Q, $\Delta_Q = \Delta_Q[D, \Delta D]$ will also be interpreted as an annotated relation, called the *intervention mapping of* Q.

4.2.1 Addition/Subtraction on Annotated Relations

Two annotated relations S and S' must have the same type to be compatible for addition or subtraction.

DEFINITION 4.3. *Let* S *and* S' *be two annotated relations with type* $[\bar{\mathtt{A}}; \bar{\mathtt{K}}]$; $|\bar{\mathtt{A}}| = m$, $|\bar{\mathtt{K}}| = \ell$. *Then* $S \oplus S'$ *and* $S \ominus S'$ *are annotated relations of type* $[\bar{\mathtt{A}}; \bar{\mathtt{K}}]$ *such that* $\forall \bar{t} \in \mathbb{D}^m$,

$$\begin{aligned}
(S \oplus S')(\bar{t}) &= S(\bar{t}) + S'(\bar{t}) && \text{if } \bar{\mathtt{K}} \text{ is non-trivial annotation} \\
&= S(\bar{t}) \vee S'(\bar{t}) && \text{if } \bar{\mathtt{K}} = \mathtt{K} \text{ is trivial annotation} \\
(S \ominus S')(\bar{t}) &= S(\bar{t}) - S'(\bar{t}) && \text{if } \bar{\mathtt{K}} \text{ is non-trivial annotation} \\
&= S(\bar{t}) \wedge \overline{S'(\bar{t})} && \text{if } \bar{\mathtt{K}} = \mathtt{K} \text{ is trivial annotation}
\end{aligned}$$

If S, S' *have trivial annotation, we use the following shorthand for the* intersection *of* S, S': $S \odot S' = S \ominus (S \ominus S')$.

[3]If base relations have duplicates, the tuples can be made unique by considering an additional *id* column.

[4]This particular subclass is considered for technical reasons as mentioned later in the paper (*e.g.*, see Section 4.3.1).

For relations with trivial annotations, $\oplus, \ominus, \odot$ can be implemented by SQL union, minus/except, intersect respectively. For non-trivial annotations, a tuple present in either of S, S' should be present in the output (even if it does not appear in one of the relations). Therefore, to implement $\oplus, \ominus$, we run an SQL query that performs a *full outer join* on the tables S, S' on $S.\bar{\mathtt{A}} = S'.\bar{\mathtt{A}}$, and computes the aggregate $\mathtt{isnull}(S.\alpha, 0) \pm \mathtt{isnull}(S'.\alpha, 0)$ as α for each $\alpha \in \bar{\mathtt{K}}$. Here the function $\mathtt{isnull}(A, 0)$ returns 0 if $A = null$ (when it is not present in one of the tables), otherwise returns A.

Remark. We are referring to two join operators. The algebra on annotated relations includes a join operation as defined in equation (2) to capture the standard join operation supported in relational algebra or SQL. However, the $\oplus$ and $\ominus$ operators described above will also require the standard join (outer-join) operation in SQL when they are implemented. In other words, we first interpret extended relational algebra operators as operators on annotated relations in equations (2) and (3). Then to implement these operators in a DBMS, we translate them back to SQL queries.

4.2.2 Intervention Mapping on Base Relations

Recall that the type of annotated base relation S_i is $[\bar{\mathtt{A}}_i; \mathtt{K}]$, and E denotes the integer index of possible explanations. Suppose $|\bar{A}_i| = m_i$; $|\mathtt{K}| = |E| = 1$.

Definition 4.4. *The* intervention mappings *of base relations, $\Delta_i : \mathbb{D}^{1+m_i} \to \mathbb{R}$ for $i \in [1, k]$, are annotated relations of type $[\{E\} \cup \bar{A}_i; \mathtt{K}]$ such that $\mathtt{K}$ is the trivial annotation, and $\forall E = e$, $\forall \bar{t} \in \mathbb{D}^{m_i}$, $\Delta_i(\langle e, \bar{t}\rangle) \Rightarrow S_i(\bar{t})$, i.e., if the LHS is 1, the RHS must also be 1.*

The above condition ensures that if a tuple $\bar{t} \in \mathbb{D}^m$ is deleted from S_i for an explanation $E = e$, then $\bar{t}$ must exist in S_i. Note that $\Delta_i : \mathbb{D}^{1+m_i} \to \mathbb{R}$ is equivalent to $\Delta_i : \mathbb{D} \to [\mathbb{D}^{m_i} \to \mathbb{R}]$. By selecting tuples from Δ_i where $E = e$, and discarding the attributes E, we get another annotated relation (all the values of $\bar{\mathtt{A}}_i$ are unique), which we denote by $\Delta_{i,e}$. Hence for an explanation $E = e$, the intervention of e is given by $\Delta D_e = (\Delta_{1,e}, \cdots, \Delta_{k,e})$. Further, for each explanation $E = e$, the intervened relation $\hat{S}_i$ due to e is precisely captured by the annotated relation of type $[\bar{\mathtt{A}}_i; \mathtt{K}]$: $\hat{S}_i = S_i \ominus \Delta_{i,e}$ (see Figure 1).

4.2.3 Intervention Mapping of a Query

Next, we generalize Definition 4.4 and define Δ_Q for any query Q that is defined through the algebra in (2) and (3).

Definition 4.5. *Let D be a database comprising annotated relations S_i of type $[\bar{\mathtt{A}}_i; \mathtt{K}]$, $i \in [1, k]$. Let Q be a query that takes D as input and produces an annotated relation $Q[D]$ of type $[\bar{A_Q}; \bar{K_Q}]$ as output; $|\bar{A_Q}| = m$ and $|\bar{K_Q}| = \ell$. Let E be the index of explanations and Δ_i be the intervention mapping of S_i, $i \in [1, k]$. Let $D \ominus \Delta D_e$ denote the database comprising annotated relations $S_1 \ominus \Delta_{1,e}, \cdots, S_k \ominus \Delta_{k,e}$.*

Then, the intervention-mapping *of Q, denoted by $\Delta_Q : \mathbb{D}^{1+m} \to \mathbb{R}^\ell$, equivalently $\Delta_Q : \mathbb{D} \to [\mathbb{D}^m \to \mathbb{R}^\ell]$, is an annotated relation of type $[\{E\} \cup \bar{A_Q}; \bar{K_Q}]$ such that for all $E = e$,*

$$Q[D] \ \ominus \ \Delta_{Q,e}[D, \Delta D] \ = \ Q[D \ominus \Delta D_e] \qquad (4)$$

where $\Delta_{Q,e}$ is the restriction *of ΔQ to $E = e$ that selects tuples with $E = e$ from Δ_Q and discards E.*

4.3 Computing Intervention Mapping

Now we describe how Δ_Q can be computed incrementally along any given logical query plan tree assuming Δ_i for $i \in [1, k]$ have been precomputed and stored. For a tuple $\bar{t}$, an attribute a, and a vector of attributes $\bar{b}$, $\bar{t}.a$ denotes the value of attribute a in $\bar{t}$ and $\bar{t}.\bar{b}$ denotes the vector of the values of the attributes $\bar{b}$ in $\bar{t}$.

4.3.1 For Non-Aggregate Operators

The rules for each non-aggregate operator in the grammar (2) are given below (here all the input and output annotated relations have trivial annotations). These rules are similar to the update rules in the IVM literature [12, 18], although they take into account updates for all explanations e simultaneously. The correctness of these rules follows by induction (proofs and illustrating examples appear in [4]). Here ER is a relation with attribute E that contains all possible values of explanation index $E = e$.

Base case: If $Q = S_i$, $i \in [1, k]$, return $\Delta_Q = \Delta_i$.
Selection: If $Q = \sigma_c(Q')$, return $\Delta_Q = \sigma_c(\Delta_{Q'})$.
Join: If $Q = Q_1 \bowtie_c Q_2$, return $\Delta_Q = [(Q_1 \bowtie_c \Delta_{Q_2}) \oplus (\Delta_{Q_1} \bowtie_c Q_2)]$.
Union: If $Q = Q_1 \cup Q_2$, return $\Delta_Q = [(\Delta_{Q_1} \ominus (ER \times Q_2)) \oplus (\Delta_{Q_2} \ominus (ER \times Q_1))] \oplus (\Delta_{Q_1} \odot \Delta_{Q_2})$

Remark. The proof of the rule for σ_c depends on the equality that $\sigma_c(S \ominus S') = \sigma_c S \ominus \sigma_c S'$. However, this equality does not hold if the predicate c includes annotation attributes $\bar{\mathtt{K}}$. For instance, suppose S and S' have a single tuple each: $\langle \bar{t}, 5\rangle$ and $\langle \bar{t}, 4\rangle$ and the condition c checks whether the annotation is ≥ 2. Then, in $\sigma_c(S \ominus S')$, tuple $\bar{t}$ does not exist, whereas the tuple $\bar{t}$ has annotation 1 in $\sigma_c S \ominus \sigma_c S'$. Therefore, we needed the restriction that c is only defined on $\bar{\mathtt{A}}$, which can be relaxed as discussed in Section 6.

4.3.2 For Scalar Functions

We define intervention mapping Δg of a scalar function g which measures the change in the value of g when its inputs are changed, and allows us to do further optimizations in the incremental process of constructing Δ_Q:

$$(\Delta g)(\bar{x}, \Delta\bar{x}) = g(\bar{x}) - g(\bar{x} - \Delta\bar{x}) \qquad (5)$$

If $\Delta\bar{x} = 0$, x is unchanged, and $\Delta g = 0$. For example, if $g(x)$ = a constant, then $\Delta g = 0$; if $g(x) = x$, $\Delta g = x$, etc. For some functions, no simplifications are possible, *e.g.*, $\Delta g = \min(x, y) - \min(x - \Delta x, y - \Delta y)$ if $g(x, y) = \min(x, y)$.

Remark. The scalar functions are applied on standard or annotation attributes. However, Δx can be non-zero only if x is an annotation attribute, *e.g.*, if a tuple $\bar{r}$ appears with annotation 7 in Q and as 3 in $\Delta_{Q,e}$ for some $E = e$, then its annotation in $Q \ominus \Delta_{Q,e}$ will be 7 - 3 = 4. On the other hand, the standard attributes are treated as constants in the scalar functions g, as they either entirely exist in $Q \ominus \Delta_{Q,e}$ or are entirely omitted. We will see examples illustrating this distinction in the next subsection.

4.3.3 For Aggregate Operators

Now we consider computation of the intervention mapping ΔP for an aggregate query $P = \gamma_{\bar{\mathtt{A}}_0, sum(g(\bar{B}_0)) \to x} Q$. Only for an aggregate operator γ, the input Q can have either a trivial annotation (when Q does not have an aggregate), or a non-trivial annotation (when Q itself has been generated by an

aggregate query). In this section we give a simple algorithm to compute ΔP that takes care of both these cases.

The input Q of P is of type $[\bar{\mathtt{A}}, \bar{\mathtt{K}}]$, $\bar{\mathtt{A}}_0 \subseteq \bar{\mathtt{A}}$, g is a scalar function, and $\bar{B}_0 = \bar{B}_{0A} \cup \bar{B}_{0K}$ where $\bar{B}_{0A} \subseteq \bar{A}$ and $\bar{B}_{0K} \subseteq \bar{\mathtt{K}}$. Clearly, P is of type $[\bar{\mathtt{A}}_0, x]$. Without loss of generality, we consider only one aggregate output g. By definition of ΔP in equation (4), for any $E = e$,

$$P[D] \ominus \Delta_{P,e}[D, \Delta D] = P[D \ominus \Delta D_e]$$

Fix an e and consider a tuple $\bar{t} \in \mathbb{D}^\ell$ where $|\mathtt{A}_0| = \ell$. Next we compute $\Delta_{P,e}(\bar{t})$, *i.e.*, the change in the annotation for $\bar{t}$.

For simplicity, we denote the annotation of $\bar{t}$ in $P[D]$, $P[D \ominus \Delta D_e]$, and $\Delta_{P,e}[D, \Delta D]$ as $P_D(\bar{t})$, $P_{D-\Delta D}(\bar{t})$, and $\Delta_{P_{D,\Delta D}}(\bar{t})$ respectively (similarly for Q, Δ_Q etc.). Further, we denote the indicator function to check if the annotation of a tuple is non-zero, $I[Q(\bar{t}) \neq \bar{0}]$ as $b_Q(\bar{t})$. Recall the definition of Δg (5). Since $P_D(\bar{t}) - \Delta P_{D,\Delta D}(\bar{t}) = P_{D-\Delta D}(\bar{t})$,

$$\begin{aligned}
&\Delta P_{D,\Delta D}(\bar{t}) \;=\; P_D(\bar{t}) - P_{D-\Delta D}(\bar{t}) \\
&= \sum_{\bar{r}\,:\,\bar{r}.\bar{\mathtt{A}}_0 = \bar{t}} g(\langle \bar{r}.\bar{B}_{0A}, Q_D(\bar{r}).\bar{B}_{0K}\rangle) \times b_{Q_D}(\bar{r}) \\
&\quad - g(\langle \bar{r}.\bar{B}_{0A}, Q_{D-\Delta D}(\bar{r}).\bar{B}_{0K}\rangle) \times b_{Q_{D-\Delta D}}(\bar{r}) \\
&= \sum_{\bar{r}:\bar{r}.\bar{\mathtt{A}}_0 = \bar{t}} g(\langle \bar{r}.\bar{B}_{0A}, Q_D(\bar{r}).\bar{B}_{0K}\rangle) \times b_{Q_D}(\bar{r}) \\
&\quad - g(\langle \bar{r}.\bar{B}_{0A}, Q_D(\bar{r}).\bar{B}_{0K}\rangle) \times b_{Q_{D-\Delta D}}(\bar{r}) + \quad b_{Q_{D-\Delta D}}(\bar{r}) \times \\
&\Delta g(\langle \bar{r}.\bar{B}_{0A}, Q_D(\bar{r}).\bar{B}_{0K}\rangle, \langle \bar{r}.\bar{B}_{0A}, \Delta Q_D(\bar{r}).\bar{B}_{0K}\rangle)
\end{aligned}$$

Interestingly, this complex expression can be simplified and efficiently implemented using the properties of a monotone query. Due to space constraints, here we give the algorithm; the analysis can be found in the full version [4].

Perform a semi-join $\Delta Q \ltimes_{\Delta Q.\bar{\mathtt{A}} = Q.\bar{\mathtt{A}}} Q$, where $\bar{\mathtt{A}}$ is the set of all standard attributes of Q, and compute a new attribute y as follows: **if** $Q.\bar{\mathtt{K}} = \Delta Q.\bar{\mathtt{K}}$ **then**
  / (**Case 1**): This includes the case when $\bar{\mathtt{K}}$ is the trivial annotation */*;
  $y = g(\langle Q.\bar{B}_{0A}, Q.\bar{B}_{0K}\rangle)$
end
else
  / (**Case 2**) */*;
  $y = \Delta g(\langle Q.\bar{B}_{0A}, Q.\bar{B}_{0K}\rangle, \langle Q.\bar{B}_{0A}, \Delta Q.\bar{B}_{0K}\rangle)$
end
Perform GROUP BY on $\bar{B}_{0A}$ and store $sum(y)$ as x.

Algorithm 1: Generic algorithm for computing ΔP from ΔQ, where $P = \gamma_{\bar{\mathtt{A}}_0, sum(g(\bar{B}_0)) \to x} Q$

There is a subtle difference between how standard attributes and annotation attributes are treated in the above algorithm, which we illustrate with the examples below. In the first example, there are no non-trivial annotation attributes.

EXAMPLE 4.6. ***(No non-trivial annotation in Q):*** *Consider the following intermediate relation Q and its ΔQ:*

Q

A	B	C		
a_1	b_1	9	→	1
a_1	b_2	10	→	1
a_1	b_3	5	→	1
a_2	b_1	9	→	1

Δ_Q

E	A	B	C		
1	a_1	b_1	9	→	1
1	a_1	b_3	5	→	1
1	a_2	b_1	9	→	1
2	a_1	b_1	9	→	1
2	a_1	b_2	10	→	1
3	a_2	b_1	9	→	1

Consider $P = \gamma_{A,sum(C)\to x} Q$. There are no non-trivial annotations in Q, therefore all tuples $\bar{r} \in \Delta_Q$ fall under the first case in Algorithm 1, i.e., all such tuples $\bar{r}$ do not survive in $Q_{D-\Delta D}$ and contribute the original value of the scalar function $g(C)$ to the sum y. Hence, Δ_P will be computed by Case 1 in Algorithm 1. For $E = 2$, $P \ominus \Delta_{P,2}$ is shown below, which equals to result of the a query P on $Q \ominus \Delta_{Q,2}$. Here ΔP is simply the aggregate query $\Delta_P = \gamma_{E,A,sum(C)\to x}$.

Δ_P

E	A		x
1	a_1	→	14
1	a_2	→	9
2	a_1	→	19
3	a_2	→	9

P

A		x
a_1	→	24
a_2	→	9

$Q \ominus \Delta_{Q,2}$

A	B	D		
a_1	b_3	5	→	1
a_2	b_1	9	→	1

$P[Q \ominus \Delta_{Q,2}] = P \ominus \Delta_{P,2}$

A		x	
a_1	→	5	*(= 24-19)*
a_2	→	9	*(=9-0)*

Generalizing the above example, from Algorithm 1, we get the following optimization (note that the semi-join is on $\Delta Q.\bar{\mathtt{A}} = Q.\bar{\mathtt{A}}$, since $\bar{A}_0 \subseteq \bar{\mathtt{A}}$, $\Delta Q.\bar{\mathtt{A}}_0 = Q.\bar{\mathtt{A}}_0$):

If the annotated relation has no non-trivial annotation, then for $P = \gamma_{\bar{\mathtt{A}}_0, sum(g(\bar{B}_0)) \to x} Q$:
$\Delta_P = \gamma_{E,\bar{\mathtt{A}}_0, sum(g(\bar{B}_0)) \to x} \Delta_Q$

Next we will study a slight variation of Example 4.6.

EXAMPLE 4.7. ***(Non-trivial annotation in Q):*** *Consider the following intermediate relation Q and its Δ_Q:*

Q

A	B		K
a_1	b_1	→	9
a_1	b_2	→	10
a_1	b_3	→	5
a_2	b_1	→	9

Δ_Q

E	A	B		K
1	a_1	b_1	→	5
1	a_1	b_3	→	2
1	a_2	b_1	→	9
2	a_1	b_1	→	9
2	a_1	b_2	→	4
3	a_2	b_1	→	7

Consider the same query $P = \gamma_{A,sum(K)\to x} Q$. The scalar function is $g(y) = y$, and therefore $g(y, \Delta y) = \Delta y$. Consider $E = 2$. Here the first tuple $\langle 2, a_1, b_1, 9\rangle$ falls under the first case in Algorithm 1 (the annotation of the tuple, i.e. 9, is the same in Δ_Q and Q), and the second tuple $\langle 2, a_1, b_2, 4\rangle$ falls under the second case (the annotation of the tuple is 4 is Δ_Q and 10 in Q). Hence the first tuple contributes the original value of the scalar function $g(\Delta_Q.K) = 9$ to the sum x, whereas the second tuple contributes $\Delta g(Q.K, \Delta_Q.K) = \Delta_Q.K = 4$ to the sum x. For $E = 2$, $P \ominus \Delta_{P,2}$ is shown below.

Δ_P

E	A		x
1	a_1	→	7
1	a_2	→	9
2	a_1	→	13
3	a_2	→	7

$Q \ominus \Delta_{Q,2}$

A	B		K	
a_1	b_2	→	6	*(= 10-4)*
a_1	b_3	→	5	
a_2	b_1	→	9	

P

A		x
a_1	→	24
a_2	→	9

$P[Q \ominus \Delta_{Q,2}] = P \ominus \Delta_{P,2}$

A		x	
a_1	→	11	*(= 24-13)*
a_2	→	9	*(=9-0)*

Interestingly, similar to Example 4.6, Δ_P in Example 4.7 also can be computed as $\Delta_P = \gamma_{E,A,sum(K)\to x}\Delta_Q$. However, this follows due to a completely different argument. The most common scalar function is $g(x) = x$, *i.e.*, the $x = sum(C)$ for an attribute C. If $C \in \mathtt{K}$ is an annotation attribute, then in case 1, $g = Q.C = \Delta_Q.C$, and in case 2, $\Delta g = g(Q.C) - g(Q.C - \Delta_Q.C) = \Delta_Q.C$. Note that $\Delta g(x, \Delta x) = \Delta x$ (*i.e.*, the additive nature of $sum(C)$) is important here – for other scalar functions g such that Δg involves both $x, \Delta x$ (*e.g.*, $g(x) = x^2, \min(x)$ etc.), the semi-join of Δ_Q with Q will be necessary. This gives the second optimization rule:

For $P = \gamma_{\bar{\mathtt{A}}_0,sum(C)\to x}Q$, where $C \in \mathtt{K}$ is an annotation attribute (or is a constant), $\Delta_P = \gamma_{E,\bar{\mathtt{A}}_0,sum(C)\to x}\Delta_Q$

A special case of the above rule is when $C = 1$ or other real constants, which can be treated as trivial annotations. Therefore, the above optimization can be applied for count queries like $P = \gamma_{\bar{\mathtt{A}}_0,sum(1)\to x}Q$.

Remark. The above optimization rule does not hold if $C \in \mathtt{A}$ is a standard attribute. In case 1, similar to the above, $g = Q.C = \Delta_Q.C$. However, to check if the tuple falls under case 1, the semi-join is needed. This is due to the fact that in case 2, $\Delta g = g(Q.C) - g(Q.C) = 0$ and not $\Delta_Q.C$, *i.e.*, if a tuple $\bar{r}$ survives in $Q - \Delta Q$, it contributes $Q.C$ entirely to the sum as C is a standard attribute and not an annotation attribute.

4.4 Explanation-Query from Intervention Mapping

To compute $Q[D \ominus \Delta D_e]$, we do not need to compute $Q[D] \ominus \Delta_{Q,e}[D, \Delta D]$ for each e. Instead, a left outer-join is performed between the original query answer $Q[D]$ and $\Delta_Q[D, \Delta D]$ on equal values of standard attributes $\bar{\mathtt{A}}_Q$, a group by is performed on $\{E\} \cup \bar{A}_Q$, and the updated annotations are computed by subtraction or set difference of $\Delta_Q.\bar{\mathtt{K}}$ from $Q.\bar{\mathtt{K}}$ for each such group (that will have the same value e of E) for non-trivial and trivial annotations respectively; only the non-zero results are returned. However, sometimes the difference of the original and the new query captured by the $\Delta_Q[D, \Delta D]$ relation will suffice to rank the explanations (*e.g.*, when the aggregate function is additive), and then this join at the end can be avoided. The explanation-query (all steps of Δ_Q and the final outer-join with Q if needed) is sent to the DBMS as a single query to utilize the optimizations by the DBMS. If the user question involves multiples queries, then explanation query is constructed for all of them, and the answers are combined for the final ranking of explanations by a top-k query (an example is in Section 5.2).

EXAMPLE 4.8. *The results of the final explanation query for the query P in Examples 4.6 and 4.7 are shown below (computed from $P, \Delta P$-s in Examples 4.6 and 4.7).*

E	A		x
1	a_1	→	10
2	a_1	→	5
2	a_2	→	9
3	a_1	→	24

$P[D - \Delta D]$ ***in Ex.4.6***

E	A		x
1	a_1	→	17
2	a_1	→	11
2	a_2	→	9
3	a_1	→	24
3	a_2	→	2

$P[D - \Delta D]$ ***in Ex.4.7***

5. EXPERIMENTS

We present experiments to evaluate our framework in this section. The prototype of our system is built in Java with JDK 6.0 over Microsoft SQLServer 2008. The input query is parsed with JSqlParser [8] and a basic query plan is generated. The explanation query is constructed along this plan and is sent to the database system. All experiments were run locally on a 64-bit Windows Server 2008 R2 Datacenter with Intel(R) Xeon(R) 4-core X5355 processor (16 GB RAM, 2.66 GHz).

5.1 Running Time Evaluation

The input consists of relations R_i with attributes $\bar{\mathtt{A}}_i$, $i \in [1, k]$, corresponding Δ_i with attributes $\{E\} \cup \bar{\mathtt{A}}_i$, and a query Q. The goal is to compute the new value of the query for each explanation $e \in E$ when $\Delta_{i,e} = \pi_{\bar{A}_i}\sigma_{E=e}\Delta_i$ is removed from R_i. Our algorithm is referred to as `SingleQ-IVM` in the figures, and is compared with the runtime of the original query Q (called `OrigQ`) that computes $Q[D]$. In addition, we compare `SingleQ-IVM` against the following alternative approaches in Section 5.1.1. Comparisons with DBToaster[9, 3] is provided in Section 5.1.2.

1. (`Naive-Iter`) Iterate over all $e \in E$. For each e, evaluate the query on relations $R_i - \pi_{\bar{A}_i}\sigma_{E=e}\Delta_i$.
2. (`Naive-SingleQ`) Let `Expl`(E) store all explanations $E = e$. Then evaluate the query on relations $(\mathtt{Expl} \times R_i) - \Delta_i$, $i \in [k]$. Here only one query is issued (the iteration over $E = e$ is not needed), but expensive cross products of the given relations with `Expl` are performed.
3. (`Iter-IVM`) For each $E = e$, implement IVM using rules similar to those in Section 4.3.1 after computing $\Delta_{i,e} = \pi_{\bar{A}_i}\sigma_{E=e}\Delta_i$. The original query Q is evaluated once and then the new values $Q[D - \Delta D_e]$-s are computed incrementally by IVM for all e. This helps us compare against IVM approaches that go over all e sequentially, and illustrates the advantage of running a single incremental query for all e.

5.1.1 Dependency of on Different Parameters

A number of parameters affect the running time of the algorithms: (i) the total number of tuples in the database; (ii) the number of explanations $|E|$; (iii) the number of tuples in each ΔD_e (referred to as *group size*); this can be different for different explanations, and we consider two methods: (a) when each explanation e has the same number of tuples, and (b) *Rand-x*: when the number of tuples for an explanation e is a random integer from 1 to x; (iv) whether the new values of the query answers, $Q[D - \Delta D_e]$, are sought as the output, or whether only the changes in the query answers in $\Delta_Q[D]$ suffice (see Section 4.4); we will refer to our algorithm for the latter as `SingleQ-IVM-Diff` and the corresponding version for `Iter-IVM` as `Iter-IVM-Diff`; (v) how complex the input query is; and (vi) optimizations described in the previous sections.

We evaluate dependencies of the running time of our algorithms `SingleQ-IVM` and `SingleQ-IVM-Diff` using synthetic ΔD relations generated from *NCHS' Vital Statistics Natality Birth Data, 2010* [5]; real ΔD-s are considered in the next section for more complex queries. This dataset comprises a single table (*Natality*) with more than 4M anonymized tuples, each having 233 attributes. As the tuples are independent, the ΔD relations can be generated by randomly

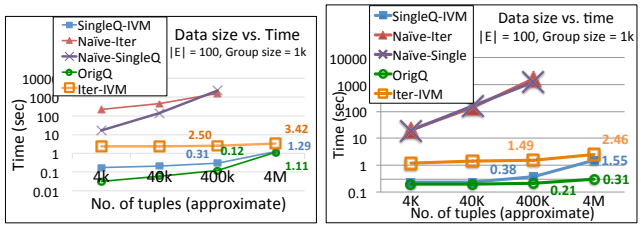
choosing a subset for different values of the above parameters. We consider two queries. **(A)** The first query is a multi-output group-by query $\mathbf{Q_{GB}}$ that counts the number of births at different *APGAR scores* (*AP*), a measure for the health of the baby five minutes after birth:

```
SELECT AP, COUNT(*) FROM Natality GROUP BY AP
```

(B) The second query $\mathbf{Q_M}$ targets a user question mentioned in [32]: *'why the number of babies with high APGAR score* $\in [7, 10]$ *is high if the mother is married'* (the attribute MaritalStatus is 1), and returns a single number as output:

```
SELECT COUNT(*) FROM Natality
WHERE MaritalStatus = 1 and AP >= 7 and AP <= 10
```

For both queries, Δ_Q is constructed and evaluated using the optimization in Section 4.3.3 which is returned for `SingleQ-IVM-Diff`. For `SingleQ-IVM`, the final answers $Q[D - \Delta D]$ requires computation of $Q[D]$ and a left outer join between $Q[D]$ and $\Delta_Q[D, \Delta D]$. Q_{GB} computes a group-by and operates on the entire table, whereas Q_M selects a subset of tuples by its selection condition and computes a single number. Therefore, although ΔQ_M has an additional step for selection in the query plan for Q_M, in all the graphs, Q_M takes less time than Q_{GB}.

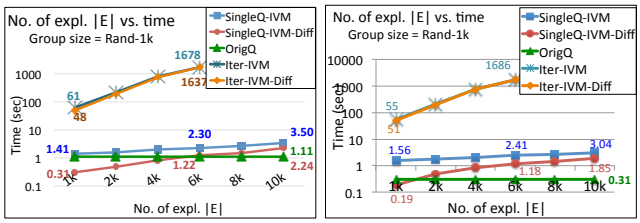


(a) For query Q_{GB} (b) For query Q_M

Figure 2: Data size vs. time (logscale), $|E|$ = 100, grp. size = 1k

(i) **Data size with time.** First, we vary the number of tuples in the base relations in multiples of 10: 4k, 40k, 400k, and 4M. For each of these four base relations, Δ relations are created by randomly choosing 1000 tuples (group size) 100 times (= $|E|$). In Figures 2a and 2b, our algorithm `SingleQ-IVM` is compared with the original query `OrigQ`, and the other alternative approaches for Q_{GB} and Q_M respectively. In this figure, and also for most of the data points in the subsequent Figures 3 and 4, the difference in running times of `SingleQ-IVM` and `OrigQ` is < 3 sec. Moreover, it outperforms `Iter-IVM` that also runs IVM but iterates over all 100 explanations; Figure 3 shows that the benefit is much higher for larger $|E|$-s. The other approaches `Naive-Iter` and `Naive-SingleQ` are clearly not useful (for 4M tuples they take > 3 hours and are not shown in the graphs). Using the optimization described in Section 4.3.3 for relations with trivial annotation, the Δ_Q queries are evaluated on ΔD, therefore the explanation query is mostly unaffected by D (apart from the final join where $Q[D]$ has to be computed); so, the running time is not much dependent on the input data size.

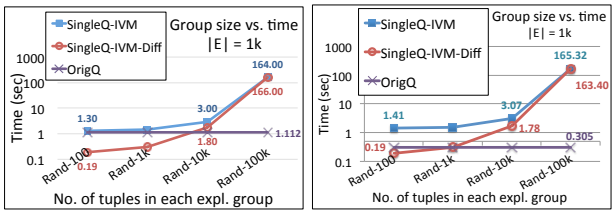
(ii) $|E|$ **vs. time.** In Figures 3a and 3b, our algorithms `SingleQ-IVM` and `SingleQ-IVM-Diff` substantially outperform `Iter-IVM` and `Iter-IVM-Diff` whose running times rapidly increase with $|E|$ (the dependency is not exactly linear as `Iter-IVM` has a fixed cost for evaluating Q, and the sizes of ΔD_e are chosen at random and therefore are not



(a) Q_{GB}, grp. size = Rand-1k (b) Q_M, grp size = Rand-1k

Figure 3: No. of explanations $|E|$ vs. time (logscale), $|D|$ = $4M$

uniform). The increase in running times of `SingleQ-IVM` and `SingleQ-IVM-Diff` with $|E|$ is much slower, although `SingleQ-IVM` does not give an interactive performance when both $|E|$ and all $|\Delta D_e|$-s are large (*e.g.*, the running time is 50 sec when $|E|$ = $4k$, group size = Rand-10k, and $|\Delta D| \approx 20M$). However, in practice, not all groups are likely to have a large number of tuples as suggested by the real ΔD_e-s constructed in the following section, leading to a running time of a few seconds in most of our actual queries. The difference in time for `SingleQ-IVM` and `SingleQ-IVM-Diff` is small, implying that the final join of $\Delta_Q[D]$ with $Q[D]$ is not the main source of complexity. Moreover, `SingleQ-IVM-Diff` can even outperform the original query Q, when ΔD, even considering all E = e, still has fewer tuples than D.



(a) For query Q_{GB} (b) For query Q_M

Figure 4: Group size vs. time (logscale), $|E|$ = $1k$

(iii) **Group size vs. time.** In Figures 4a and 4b, $|E|$ = $1k$, $|D|$ = $4M$, and the group size is varied at a multiple of 10, starting from Rand-100 up to Rand-100k for Q_{GB} and Q_M respectively. The total sizes of ΔD relations at these four points are respectively about 50k, 514k, 5.1M, and 50M (whereas the size of D is 4M). The algorithm does not have an interactive speed at Rand-100k (166 sec), but such high values of ΔD_e for all e are less likely in practice. If the optimizations in Section 4.3.3 are not used, the running time for Rand-10k and Rand-100k in Figure 4a increases by about 20 sec, which is predominantly due to the equi-join of D with ΔD on 233 attributes.

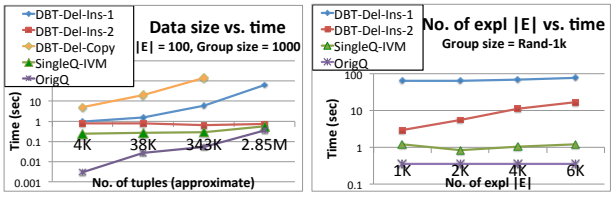
5.1.2 *Comparisons with DBToaster*

Next, we compare our algorithm `SingleQ-IVM` that computes $Q[D - \Delta D_e]$ for all explanations e simultaneously with DBToaster[9, 3], a state-of-the-art framework for IVM. These results are presented separately since DBToaster actually performs database updates by inserting or deleting tuples, in contrast to the alternative approaches discussed above, none of which modify the database. We incorporated the compiled JAR file of the Scala code for the query Q_M from a Java program that used Akka Actors

[6, 3] to submit individual events to insert/delete tuples[5] . DBToaster 'reads' the database from a file, so we exported the databases from SQLServer to csv files, which were read by the Java program[6] We considered two approaches to compute $Q[D - \Delta D_e]$ sequentially for all e using DBToaster:

- (`DBT-Del-Ins`) Read database D. For each e, (a) delete all tuples in Δ_e one by one, (b) take the snapshot of $Q[D - \Delta_e]$, (c) Insert all tuples in Δ_e back to D one by one so that the database is ready for the next e. In Figure 5, `DBT-Del-Ins-1` takes into account the time to read D from the csv file, and `DBT-Del-Ins-2` ignores it.
- (`DBT-Del-Copy`) For each e, (a) read database D (*i.e.*, get a new copy of D), (b) delete all tuples in Δ_e one by one.

The first approach does not require reading the database D multiple times, but needs to insert back the tuples in Δ_e. We did not take into account the time to compile the SQL code into a Scala program and a JAR (a few seconds).



(a) Data size vs. time (logscale) (b) No. of explanations $|E|$ vs. time (logscale), $|D| = 2.85M$

Figure 5: Comparisons with DBToaster, query = Q_M

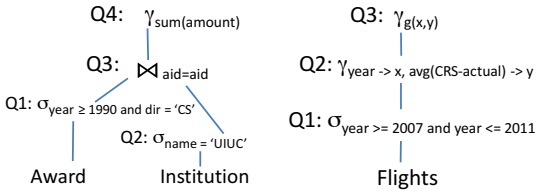
Figure 5a varies the number of tuples in the database and compares the above three DBToaster simulations with our algorithm `SingleQ-IVM` and with `OrigQ` that computes $Q[D]$. Our algorithm `SingleQ-IVM` outperforms both DBToaster approaches, even when the time to read the database D is not taken into account (`DBT-Del-Ins-2`). `DBT-Del-Copy` is not scalable (it took more than 27 mins for 2.85M tuples in the first plot, and therefore is omitted in the second plot). `DBT-Del-Ins` performs better, although does not give interactive performances even for `DBT-Del-Ins-2` when the number of explanations is large (it took 16.9 sec for $|E| = 6k$ in Figure 5b, whereas our `SingleQ-IVM` algorithm took 1.2 sec).

5.2 Complex Queries

Now we consider queries involving joins and multi-level aggregates with real ΔD datasets. The first experiment focuses on Example 1.1 in the introduction. The number of tuples in *Award*, *Institution*, and *Investigator* are respectively about 400k, 419k, and 621k (the first two tables are used in the query). The total number of explanations is more than 188k, and the total number of tuples in the Δ_{Award} table is more than 1.74M. The Δ_{Award} table is populated using eight query templates that perform join/nested aggregates on the *Award*, *Investigator*, and *Institution* tables. The actual explanations with their sub-ids are stored in seven separate tables (*e.g.*, for *Type-1*, the *sub-id* and the *PIName*

[5]The latest released version of DBToaster does not support batch updates (from personal communication with the DBToaster team).

[6]The numbers of tuples in Figures 5a and 2b are different since DBToaster did not allow 233 attributes of the Natality database and we projected the tables on 21 of the attributes.



(a) NSF awards (Q_u) (b) Flight data

Figure 6: Query plans for incremental computation of Δ_Q-s

are stored). These explanations and their interventions are computed before any query is issued or any user question is asked; therefore some of the explanations involve schools other than `UIUC` or `CMU`, and may not affect the user question.

We consider the first seven types of the complex explanations as shown in Table 1 for simplicity to explain the question in Example 1.1. The interventions of these explanation types only affect the *Award* table and not the *Institution* table (if a PI did not exist, his/her institution still remains unaffected), therefore $\Delta_{Institution} = \varnothing$. The *Investigator* table does get affected for some explanations but we do not discuss it here as it is not used in the query in Example 1.1. The id of the explanation E comprises the *type* and a *sub-id* within the type.

Given the answer to the group by query in Example 1.1, the user question was why the difference of the amounts for `UIUC` and `CMU` was high, *i.e.*, why $Q_u[D] - Q_c[D]$ is high, where Q_u, Q_c are two single-output (no group-by) aggregate queries that compute the total award amount for `UIUC` and `CMU` respectively. A good explanation e will make the difference low, *i.e.*, $Q_u[D - \Delta D_e] - Q_c[D - \Delta D_e]$ will be low, where ΔD_e is the intervention of e. Since the aggregate function sum is additive, it suffices to compute the differences $\Delta_{Q_u}[D, \Delta D]$ and $\Delta_{Q_c}[D, \Delta D]$, which are computed using rules in Section 4.3 along the query plan in Figure 6a (the plan for Q_c is similar). After the results of $\Delta_{Q_u}, \Delta_{Q_c}$ are computed, these two relations are combined using a full outer join (so that we do not lose any explanation that only appears in one of the two), and the top explanations e according to the highest values of $\Delta_{Q_u,e}[D, \Delta D] - \Delta_{Q_c,e}[D, \Delta D]$ are returned. The computation of Δ_{Q_u} and Δ_{Q_c} takes about 0.8 sec each, and the final answers are obtained within 1.8 sec.

We found the following as top explanations: (i) PIs with more than 100M awards in total (there were none in `CMU`), (ii) PIs with $\geq$ 11 and 7-8 awards, (iii, iv) particular PIs from `UIUC` '*Robert Pennington*' and '*Thomas Dunning*' (who share some awards), (v) awards with 6 or more PIs. Each of these explanations contributes to more than \$400M difference of award amounts of `UIUC` and `CMU` going up to $\geq$ \$850M. The above explanations are diverse and give the insight that a few PIs in `UIUC` got a number of big awards which contribute the most toward the difference, which could not be obtained if we were restricted to simple predicates from the *Award* and *Institution* tables.

The second experiment is on the *flights* dataset [7], the on-time arrival records for flights in the USA in the month of December from 1987 to 2012 (a single table with about 12.63M tuples and 22 attributes). We give an overview of this experiment due to space constraints (details in [4]). The query we had considered asks why the *slope* of a linegraph

(using linear regression) is high and is given below:

```
WITH R AS
 (SELECT YEAR as x, AVG(CRS_TIME - ACTUAL_TIME) as y
  FROM Flights
  WHERE YEAR >= 2007 and YEAR <= 2011
  GROUP BY YEAR )
SELECT ((COUNT(R.x)*SUM(R.x*R.y))-(SUM(R.x)*SUM(R.y)))/
 ((count(R.X)*(SUM(R.x*R.x))-(SUM(R.x))*(SUM(R.x))))
FROM R
```

Such nested queries are not supported by the on-the-fly approaches in [34, 32]. A query plan for the above query is shown in Figure 6b, where g(x,y) denotes the aggregate function that computes the slope. The relation $\Delta_{Flights}$ contained 887 explanations ($|E|$ = 887) and 138M tuples in total. The time taken is 82 sec for both `SingleQ-IVM` and `SingleQ-IVM-Diff`; the time taken to compute the original query is 10 sec. For another instance of $\Delta Flights$ with about 100M tuples, the time taken is about 61 sec. Improving the space requirement and running time for such larger datasets is an interesting direction for future work.

6. DISCUSSIONS AND FUTURE WORK

Our framework can be extended further to support larger classes of inputs. In Section 3, we assumed that all $\sigma, \bowtie, \cup$ operators appear below all aggregate operators γ in the query plan. However, this condition was only to ensure that Δ_Q can be computed incrementally along the query plan. In particular, once the relation $Q[D - \Delta D]$ is computed from D and ΔD (as described in Section 4.2.3), we can perform selection (even on annotation attributes) and join on the answer directly. If an aggregate is performed on an aggregate sub-query like $\min / \max /$*count distinct*$/avg$, then also $Q[D-\Delta D]$ has to be computed before any further operation. An interesting direction is to support unions on aggregates: *e.g.*, SQL allows union of aggregate and non-aggregate attributes with the same name, but they will behave very differently in Δ_Q relations. Optimizations are possible when the interventions of a set of explanations $e_1, \cdots, e_n$ follow $\Delta D_{e_1} \subseteq \cdots \subseteq \Delta D_{e_n}$ (*e.g.*, explanations of top-k form where k varies); in this case we only need to store and compute for the differences using rules similar to those in Section 4.3.1. Other related questions are supporting SQL queries with bag semantics, negation, and nulls, and improving the space/time requirements for larger datasets and for relatively simpler explanations like predicates spanning multiple tables so that they can be evaluated for complex nested aggregate queries. Finally, finding rich, complex explanations from datasets with minimal manual help remains an open-ended future work.

Acknowledgement. We thank the anonymous reviewers for their detailed feedback, and Milos Nikolic for helpful communication about DBToaster. This work was supported in part by NSF grant IIS-0911036.

7. REFERENCES

[1] *http://www.nsf.gov/awardsearch/download.jsp*.
[2] *http://grad-schools.usnews.rankingsandreviews.com/best-graduate-schools/top-science-schools/computer-science-rankings*.
[3] *http://www.dbtoaster.org/*.
[4] *https://users.cs.duke.edu/~sudeepa/ExplReady-FullVersion.pdf*.
[5] *http://www.cdc.gov/nchs/data_access/ftp_data.htm*.
[6] *http://akka.io*.
[7] *http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time*.
[8] Jsqlparser. *http://jsqlparser.sourceforge.net*.
[9] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *PVLDB*, 5(10):968–979, 2012.
[10] Y. Amsterdamer, D. Deutch, and V. Tannen. Provenance for aggregate queries. In *PODS*, pages 153–164, 2011.
[11] G. Bender, L. Kot, and J. Gehrke. Explainable security for relational databases. In *SIGMOD*, pages 1411–1422, 2014.
[12] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *ACM SIGMOD Record*, 15(2):61–71, 1986.
[13] R. Chirkova and J. Yang. Materialized views. *Databases*, 4(4):295–405, 2011.
[14] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD*, pages 469–480, 1996.
[15] M. Das, S. Amer-Yahia, G. Das, and C. Yu. Mri: Meaningful interpretations of collaborative ratings. *PVLDB*, 4(11):1063–1074, 2011.
[16] A. Das Sarma, A. Parameswaran, H. Garcia-Molina, and J. Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.
[17] D. Fabbri and K. LeFevre. Explanation-based auditing. *PVLDB*, 5(1):1–12, 2011.
[18] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *ACM SIGMOD Record*, 24(2):328–339, 1995.
[19] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.
[20] Y. Katsis, K. W. Ong, Y. Papakonstantinou, and K. K. Zhao. Utilizing ids to accelerate incremental view maintenance. In *SIGMOD*, pages 1985–2000, 2015.
[21] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: debugging mapreduce job performance. *PVLDB*, 5(7):598–609, 2012.
[22] B. Kimelfeld, J. Vondrák, and D. P. Woodruff. Multi-tuple deletion propagation: Approximations and complexity. *PVLDB*, 6(13):1558–1569, 2013.
[23] C. Koch. Incremental query evaluation in a ring of databases. In *PODS*, pages 87–98, 2010.
[24] V. Le and S. Gulwani. Flashextract: A framework for data extraction by examples. In *PLDI*, pages 542–553, 2014.
[25] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.
[26] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *SIGMOD*, pages 505–516, 2011.
[27] A. Meliou, S. Roy, and D. Suciu. Causality and explanations in databases. *PVLDB*, 7(13):1715–1716, 2014.
[28] V. Nigam, L. Jia, B. Thau Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. In *PPDP*, pages 125–136, 2011.
[29] J. Pearl. *Causality: models, reasoning, and inference*. Cambridge University Press, 2000.
[30] D. Quass and J. Widom. On-line warehouse view maintenance. In *SIGMOD*, pages 393–404, 1997.
[31] N. Roussopoulos. An incremental access method for viewcache: Concept, algorithms, and cost analysis. *ACM TODS*, 16(3):535–563, Sept. 1991.
[32] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.
[33] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.
[34] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8), 2013.