Scorpion: Explaining Away Outliers in Aggregate Queries

Eugene Wu
sirrice@csail.mit.edu

Samuel Madden
madden@csail.mit.edu

ABSTRACT

Database users commonly explore large data sets by running aggregate queries that project the data down to a smaller number of points and dimensions, and visualizing the results. Often, such visualizations will reveal *outliers* that correspond to errors or surprising features of the input data set. Unfortunately, databases and visualization systems do not provide a way to work backwards from an outlier point to the *common properties* of the (possibly many) unaggregated input tuples that correspond to that outlier. We propose Scorpion, a system that takes a set of user-specified outlier points in an aggregate query result as input and finds predicates that *explain* the outliers in terms of properties of the input tuples that are used to compute the selected outlier results. Specifically, this explanation identifies predicates that, when applied to the input data, cause the outliers to disappear from the output. To find such predicates, we develop a notion of *influence* of a predicate on a given output, and design several algorithms that efficiently search for maximum influence predicates over the input data. We show that these algorithms can quickly find outliers in two real data sets (from a sensor deployment and a campaign finance data set), and run orders of magnitude faster than a naive search algorithm while providing comparable quality on a synthetic data set.

1. INTRODUCTION

Working with data commonly involves *exploratory analysis*, where users try to understand trends and general patterns by fitting models or aggregating data. Such analyses will often reveal *outliers* — aggregate values, or subgroups of points that behave differently than others. Although a multitude of tools are effective at highlighting outliers, they generally lack facilities to explain *why* a given set of outputs are outliers.

We believe *why-analysis* – describing the common properties of the input data points or tuples that caused the outlier outputs – is essential for problem diagnosis and to improve model quality. For example, Figure 1 shows a visualization of data from the Intel Sensor Data Set[1]. Here, each point represents an aggregate (either mean or standard deviation) of data over an hour from 61 sensor motes. Observe that the standard deviation fluctuates heavily (Region 1) and that the temperature stops oscillating (Region 2). Our goal is to describe the properties of the data that generated these highlighted outputs that "explain" why they are outliers. Specifically, we want

[1] http://db.csail.mit.edu/labdata/labdata.html

Proceedings of the VLDB Endowment, Vol. 6, No. 8

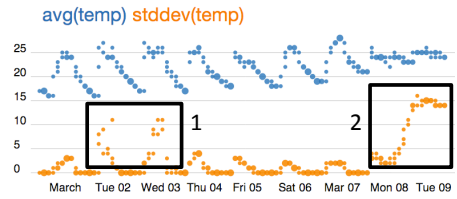

Figure 1: Mean and standard deviation of temperature readings from Intel sensor dataset.

to find a boolean predicate that when applied to the input data set (before the aggregation is computed), will cause these outliers to look normal, while having minimal effect on the points that the user indicates are normal.

In this case, it turns out that Region 1 is due to sensors near windows that heat up under the sun around noon, and the Region 2 is by another sensor running out of energy (indicated by low voltage) that starts producing erroneous readings. However, these facts are not obvious from the visualization and require manual inspection of the attributes of the readings that contribute to the outliers to determine what is going on. We need tools that can automate analyses to determine e.g., that an outlier value is correlated to the location or voltage of the sensors that contributed to it.

This problem is fundamentally challenging because a given outlier aggregate may depend on an arbitrary number and combination of input data tuples, and requires solving several problems:

Backwards provenance: We need to work backwards from each aggregate point in the outlier set to the input tuples used to compute it. In this work we assume that input and output data sets are relations, and that outputs are generated by SQL group-by queries (possibly involving user-defined aggregates) over the input. In general, every output data point may depend on an arbitrary subset of the inputs, making tracking these relationships very difficult [19].

Responsible subset: For each outlier aggregate point, we need a way to determine which subset of its input tuples cause the value to be an outlier. This problem, in particular, is difficult because the naive approach involves iterating over all possible subsets of the input tuples used to compute an outlier aggregate value.

Predicate generation: Ultimately, we want to construct a predicate over the input attributes that filter out the points in the responsible subset without removing a large number of other, incidental data points. Thus, the responsible subset must be composed in conjunction with creating the predicates.

In this paper, we describe Scorpion, a system we have built to solve these problems. Scorpion uses *sensitivity analysis* [14] to identify the groups of input points that *most influence* the outlier aggregate outputs and generate a descriptive predicate. Scorpion's problem formulation and system is designed to work with arbitrary user-defined aggregates, albeit slowly for black-box aggregates. We additionally describe properties common to many common aggregate functions that enable more efficient algorithms that extend classical regression tree and subspace clustering algorithms.

In summary, our contributions are as follows:

Tuple id	Time	SensorID	Voltage	Humidity	Temp.
T1	11AM	1	2.64	0.4	34
T2	11AM	2	2.65	0.5	35
T3	11AM	3	2.63	0.4	35
T4	12PM	1	2.7	0.3	35
T5	12PM	2	2.7	0.5	35
T6	12PM	3	2.3	0.4	100
T7	1PM	1	2.7	0.3	35
T8	1PM	2	2.7	0.5	35
T9	1PM	3	2.3	0.5	80

Table 1: Example tuples from sensors table

Result id	Time	AVG(temp)	Label	v
α_1	11AM	34.6	Hold-out	-
α_2	12PM	56.6	Outlier	$< -1 >$
α_3	1PM	50	Outlier	$< -1 >$

Table 2: Query results (left) and user annotations (right)

1. We describe several real applications, such as outlier explanation, cost analysis, enabling end-user analysts.
2. We formalize a notion of *influence* over predicates and present a system that searches for influential predicates.
3. We present common properties (similar to distributive and algebraic properties of OLAP aggregates) that enable more efficient algorithms and develop several such algorithms.
4. We run experiments on synthetic and real-world problems, showing that our algorithms are of comparable quality to a naive exhaustive algorithm while taking orders of magnitude less time to run.

The rest of the paper is structured as follows: Section 2 describes several motivating use cases and the running example used throughout the paper. Section 3 describes the formal problem formulation, Sections 4-7 present several algorithms and optimizations and experiments are described in Section 8.

2. MOTIVATION AND USE CASE

Scorpion is designed to augment data exploration tools with explanatory facilities that find attributes of an input data set correlated with parts of the dataset causing user-perceived outliers. In this section, we first set up the running example used throughout the paper, then describe several motivating use cases.

Sensor Data: Our running example is based on the Intel sensor deployment application described in the Introduction. Consider a data analyst that is exploring a sensor dataset shown in Table 1. Each tuple corresponds to a sensor reading, and includes the timestamp, and the values of several sensors. The following query groups the readings by the hour and computes the mean temperature. The left-side columns in Table 2 lists the query results.

$$\text{SELECT avg(temp), time} \quad \text{(Q1)}$$
$$\text{FROM sensors GROUP BY time}$$

The analyst thinks that the average temperature at 12PM and 1PM are unexpectedly high and wants to understand why. There are a number of questions she may want to ask that are in this vein:

1. Which of the sensors "caused" the anomalies?
2. Which sensor's values *most* "caused" the anomalies?
3. Why are these sensors reporting high temperature?
4. This problem didn't happen yesterday. What has changed?

In each of the questions, the analyst is interested in properties of the readings (e.g., sensor id) that most influenced the outlier results. Some of the questions (1 and 2) involve the *degree* of influence, while others involve comparisons between outlier results and normal results (4). Section 3 formalizes these notions.

Medical Cost Analysis: We are currently working with a major hospital (details anonymized) to help analyze opportunities for cost savings. They observed that amongst a population of cancer patients, the top 15% of patients by cost represented more than 50% of the total dollars spent. Surprisingly these patients were not significantly sicker, and did not have significantly better or worse outcomes than the median-cost patient. Their dataset consisted of a table with one row per patient visit, and 45 columns that describe patient demographics, diagnoses, a break-down of the costs, and other attributes describing the visit. They manually picked and analyzed a handful of dimensions (e.g., type of treatment, type of service) and isolated the source of cost overruns to a large number of additional chemotherapy and radiation treatments given to the most expensive patients. They later found that a small number of doctors were over-prescribing these procedures, which were presumably not necessary because the outcomes didn't improve.

Notation	Description
D	The input relational table with attributes $attr_1, \cdots, attr_k$
A_{gb}, A_{agg}	Set of attributes referenced in GROUPBY and aggregation clause
$p_i \prec_D p_j$	Result set of p_i is a subset of p_j when applied to D
α	The set of aggregate result tuples, α_i's
g_{α_i}	Tuples in D used to compute α_i e.g., have same GROUPBY key
O, H	Subset of α in outlier and hold-out set, respectively
v_{α_i}	Normalized error vector for result α_i

Table 3: Notations used

Note that simply finding individually expensive treatments would be insufficient because those treatments may not be related to each other. The hospital is interested in descriptions of high cost areas that can be targeted for cost-cutting and predicates are a form of such descriptions.

Election Campaign Expenses: In our experiments, we use a campaign expenses dataset [2] that contains all campaign expenses between January 2011 and July 2012 during the 2012 US Presidential Election. In an election that spent an unprecedented $6 billion, many people are interested in where the money was spent. While technically capable users are able to programmatically analyze the data, end-users are limited to interacting with pre-made visualizations – a *consumer* role – despite being able to ask valuable domain-specific questions about expense anomalies, simply due to their lack of technical expertise. Scorpion is a step towards bridging this gap by automating common analysis procedures and allowing end-users to perform *analyst* operations.

Extending Provenance Functionality: A key provenance use case is to trace an anomalous result backward through a workflow to the inputs that directly affected that result. A user may want to perform this action when she sees an anomalous output value. Unfortunately, when tracing the inputs of an aggregate result, the existing provenance system will flag a significant portion of the dataset as the provenance [3]. Although this is technically correct, the results are not *informative*. Scorpion can reduce the provenance of aggregate operators to a small set of influential inputs.

3. PROBLEM STATEMENT

Scorpion seeks to find a predicate over an input dataset that most influences a user selected set of query outputs. In order to reason about such tuples, we must define influence, and the type of additional information that the user can specify. Table 3 lists the notations used.

3.1 Setup

Consider a single relation D, with attributes $A = attr_1, .., attr_k$. Let Q be a group-by SQL query grouped by attributes $A_{gb} \subset A$, with a single aggregate function, $agg()$, that computes a result using aggregate attributes $A_{agg} \subset A$ from each tuple, where

[2] http://www.fec.gov/disclosurep/PDownload.do

$A_{agg} \cap A_{gb} = \emptyset$. Finally, let $A_{rest} = A - A_{gb} - A_{agg}$ be the attributes not involved with the aggregate function nor the group by that are used to construct the explanations. We model join queries by materializing the join result and assigning it as D. The predicates are then constructed with respect to the join result.

For example, Q1 contains a single group-by attribute, $A_{gb} = \{time\}$, and an aggregate attribute, $A_{agg} = \{temp\}$. The user is interested in combinations of $A_{rest} = \{SensorID, Voltage\}$ values that are responsible for the anomalous average temperatures.

Scorpion outputs the predicate that most influences a set of output results. A predicate, p, is a conjunction of range clauses over the continuous attributes and set containment clauses over the discrete attributes, where each attribute is present in at most one clause. Let $p(D) \subseteq D$ be the set of tuples in D that satisfy p. A predicate p_i is contained in p_j with respect to a dataset D if the tuples in D that satisfy p_i are a subset of those satisfying p_j: $p_i \prec_D p_j \leftrightarrow p_i(D) \subset p_j(D)$. Let P_A be the space of all possible predicates over the attributes in A.

Let the query generate n aggregate result tuples, $\alpha = \{\alpha_1, .., \alpha_n\}$, and the term *input group* ($g_{\alpha_i} \subseteq D$) be the subset of the input tuples that generate output result α_i. The output attribute, $\alpha_i.res = agg(\pi_{A_{agg}} g_{\alpha_i})$, is the result of the aggregate function computed over the projected attributes, A_{agg}, of the tuples in g_{α_i}.

Let $O = \{o_1, .., o_{n_s} | o_i \in \alpha\}$, be a subset of the results that the user flags as outliers, and $H = \{h_1, \cdots, h_{n_h} | h_i \in \alpha\}$ be a hold-out set of the results that the user finds normal. O and H are typically specified through a visualization interface, and $H \cap O = \emptyset$. Let $g_{\mathcal{X}} = \cup_{x \in \mathcal{X}} g_x | \mathcal{X} \subseteq \alpha$ be shorthand for the union of the input groups of a subset of the results, $\mathcal{X}$. For example, g_O denotes the union of the outliers' input sets.

The user can also specify an error vector that describes how an outlier result looks wrong (e.g., temperature is too high). v_{o_i} is a normalized error vector for the result, o_i that points in the direction of the error. For example, if the user thinks that α_1 of Q1 is too low, she can define the vector v_{α_1}=<-1>, whereas she can specify that α_2 is too high using v_{α_2}=<1>. Let $V = \{v_{o_i} | o_i \in O\}$, be the set of error vectors of all of the outlier results.

3.2 Predicate Influence

We will first define the influence of a predicate, p, on a single output result, o, then incorporate a user defined error vector, v_o, and finally generalize the definition to outlier and hold-out results.

Basic Definition: Our notion of influence is derived from sensitivity analysis [14], which computes the sensitivity of a model to its inputs. Given a function, $y = f(x_1, \cdots, x_n)$, the influence of x_i is defined by the partial derivative, $\frac{\Delta y}{\Delta x_i}$, which describes how the output changes given a change in x_i.

In our context, the model is an aggregate function, agg, that takes a *set* of tuples, g_o, as input, and outputs a result, o. The influence of a predicate, p, on o depends on the difference between the original result $o.res$ and the updated output after deleting $p(g_o)$ from g_o[3].

$$\Delta_{agg}(o, p) = agg(g_o) - agg(g_o - p(g_o))$$

We describe Δ_{agg} as a scalar, but it can return a vector if the agg does so. The influence is defined as the ratio between the change in the output and the number of tuples that satisfy the predicate:

$$inf_{agg}(o, p) = \frac{\Delta o}{\Delta g_o} = \frac{\Delta_{agg}(o, p)}{|p(g_o)|}$$

For example, suppose the individual influences of each tuple in $g_{\alpha_2} = \{T4, T5, T6\}$, from Tables 1 and 2. Based on the above definition, removing T4 from the input group increases the output by 10.8, thus T4 (and T5) have an influence of $inf_{AVG}(\alpha_2, \{T4\}) = \frac{56.6 - 67.5}{1} = -10.8$. In contrast, T6 has an influence of 21.6. Given this definition, T6 is the most influential tuple, which makes sense, because T6.temp increases the average the most, so removing it would most reduce the output.

The reason Scorpion defines influence in the context of predicates rather than individual or sets of tuples is because individual tuples only exist within a single input group, whereas predicates are applicable to multiple input groups. We now augment inf with additional arguments to support other user inputs.

Error Vector: The previous formulation does not take into account the error vectors, i.e., whether the outliers are too high or too low. For example, if the user thinks that the average temperature was too *low*, then removing T6 would, contrary to the user's desire, further decrease the mean temperature. This intuition suggests that only the components of the basic definition that align with the error vector's direction should be considered. This is computed as the dot product between the basic definition and the error vector:

$$inf_{agg}(o, p, v_o) = inf_{agg}(o, p) \bullet v_o$$

For example, $v_{\alpha_2} =< 1 >$ means that α_2 from Q1 is too high. The influence of T6 is $< 21.6 > \bullet < 1 >= 21.6$ and T4 is -10.8. In contrast, if $v_{\alpha_2} =< -1 >$, then T6 and T4's influences would be -21.6 and 10.8, respectively, and T4 would be more influential.

Hold-out Result: As mentioned above, the user may select a hold-out result, h, that the returned predicate should not influence. Intuitively, p should be penalized if it influences the hold-out results in any way.

$$inf_{agg}(o, h, p, v_o) = \lambda inf_{agg}(o, p, v_o) - (1 - \lambda)|inf_{agg}(h, p)|$$

Where λ is a parameter that represents the importance of not changing the value of the hold-out set. We use the absolute value of $inf_{agg}(h, p)$ to penalize *any* perturbation of hold-out results.

Multiple Results: The user will often select multiple outlier results, O, and hold-out results, H. We extend the notion by averaging the influence over the outlier results and penalizing the maximum influence over the hold-out set:

$$inf_{agg}(O, H, p, V) = \lambda \frac{1}{|O|} \sum_{o \in O} inf_{agg}(o, p, v_o) - (1 - \lambda) \max_{h \in H} |inf_{agg}(h, p)|$$

We chose to use the maximum in order to provide a hard cap on the amount that a predicate can influence *any* hold-out result.

The rest of the paper uses the following short-hands when the intent is clear from the context. $inf(p)$ denotes $inf_{agg}(O, H, p, V)$; $\Delta(p)$ denotes $\Delta_{agg}(o, p)$; $inf(t)$ and $inf(T)$ denote the influence of a predicate that matches a single tuple, t, or a set of tuples, T. We also extend Δ to accept a single or set of tuples as input.

3.3 Influential Predicates Problem

The **Influential Predicates (IP) Problem** is defined as follows: Given a select-project-group-by query Q, and user inputs O, H, λ and V, find the predicate, p^*, from the set of all possible predicates, $P_{A_{rest}}$, that has the maximum influence:

$$p^* = \arg \max_{p \in P_{A_{rest}}} inf(p)$$

While Section 2 motivated the usefulness of this problem, it is not immediately obvious why this problem should be difficult. For example, if the user thinks the average temperature is too high, why not simply return the readings with the highest temperature? We now illustrate some reasons that make the *IP* problem difficult and describe solutions in the following sections.

[3] Alternative formulations, e.g., perturbing input tuple values rather than deleting inputs tuples, are also possible but not explored here.

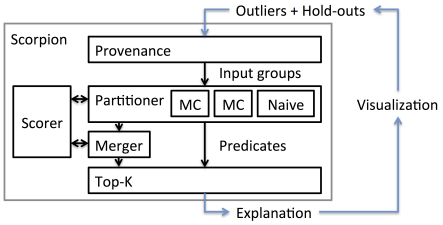

Figure 2: Scorpion architecture

The first reason is because Scorpion needs to consider how combinations of input tuples affect the outlier results, which depends on properties of the aggregate function. In the worst case, Scorpion cannot predict how combinations of input tuples interact with each other, and needs to evaluate all possible predicates (exponential in the number of and cardinalities of attributes).

The second reason is because Scorpion returns *predicates*, rather than individual tuples, to provide the user with understandable explanations of anomalies in the data. Scorpion must find tuples within bounding boxes defined by predicates, rather than arbitrary combinations of tuples. In the example above, it may be tempting to find the top-k highest temperature readings and construct a predicate from the minimum bounding box that contains those readings. However, the number of top readings is unclear, the readings may have no relation with each other, and the resulting predicate may be non-influential because the temperatures are normal or low.

The third reason is that the influence of a predicate relies on statistics of the tuples in addition to their individual influences, and the specific statistic depends on the particular aggregate function. For example, AVG depends on both the values and density of tuples, while $COUNT$ only depends on the density.

The final reason is due to the hold-out result. In the presence of a hold-out set, simple greedy algorithms may not work because an influential set of tuples in the outlier set may also influence the hold-out results.

4. BASIC ARCHITECTURE

This section outlines the Scorpion system architecture we have developed to solve the problem of finding influential predicates defined in the previous section and describes naive implementations of the main system components. These implementations do not assume anything about the aggregates so can be used on arbitrary user defined aggregates to find the most influential predicate. We then describe why these implementations are inefficient.

4.1 Scorpion Architecture

Scorpion is implemented as part of an end-to-end data exploration tool (Figure 2). Users can select databases and execute aggregate queries whose results are visualized as charts (Figure 1 shows a screenshot). Users can select arbitrary results, label them as outliers or hold-outs, select attributes that are used to construct the predicates and send the query to the Scorpion backend. Users can click through the results and plot the updated output with the outlier input tuples removed from the dataset.

Scorpion first uses the *Provenance* component to compute the provenance of the labeled results and returns their corresponding input groups. In this work, the queries are group-by queries over a single table, so computing the input groups is straightforward. More complex relationships can be established using relational provenance techniques [3]. The input groups, along with the original inputs, are passed to the *Partitioner*, which chooses the appropriate partitioning algorithm based on the properties of the aggregate. The algorithm generates a ranked list of predicates, where each predicate is tagged with a score representing its estimated influence. For example, consider the 2D dataset illustrated in Figure 3(a), where each point represents an input tuple and a darker color means higher influence. Figure 3(b) illustrates a possible set of partitions. The algorithms often generate predicates at a finer granularity than ideal (i.e., each predicate contains a subset of the optimal predicate) so they send their results to the *Merger*, which greedily merges similar predicates as long as it increases the influence (Figure 3(b)).

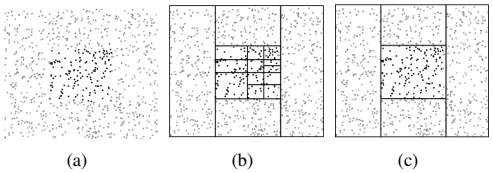

Figure 3: (a) Each point represents a tuple. Darker color means higher influence. (b) Output of *Partitioner*. (c) Output of *Merger*

The *Partitioner* and *Merger* send candidate predicates to the *Scorer*, which computes the influence as defined in the previous section. The cost is dominated by the cost of computing the Δ values, which is computed by removing the tuples that match the predicate from the original input data set and re-running the aggregate, and needs to be computed over all input groups. The need to read the entire dataset to evaluate the influence is overly expensive if the dataset is large, or the aggregate function needs multiple passes over the data. Section 5.1 describes an aggregate property that can reduce these costs. Finally, the top ranking predicate is returned to the visualization and shown to the user. We now present basic implementations of the partitioning and merging components.

4.2 Naive Partitioner (NAIVE)

For an arbitrary aggregate function without nice properties, it is difficult to improve beyond an exhaustive algorithm that enumerates and evaluates all possible predicates. This is because the influence of a given tuple may depend on the other tuples in the outlier set, so a simple greedy algorithm will not work. The NAIVE algorithm first defines all distinct single-attribute clauses, then enumerates all conjunctions of up to one clause from each attribute. The clauses over a discrete attribute, A_i, are of the form, "$\mathrm{A_i}$ in $(\cdots)$" where the $\cdots$ is replaced with all possible combinations of the attribute's distinct values. Clauses over continuous attributes are constructed by splitting the attribute's domain into a fixed number of equi-sized ranges, and enumerating all combinations of consecutive ranges. *NAIVE* computes the influence of each predicate by sending it to the *Scorer*, and returns the most influential predicate.

This algorithm is inefficient because the number of single-attribute clauses increases exponentially (quadratically) for a discrete (continuous) attribute as its cardinality increases. Additionally, the space of possible conjunctions is exponential with the number of attributes. The combination of the two issues makes the problem untenable for even small datasets. While the user can bound this search by specifying a maximum number of clauses allowed in a predicate, enumerating all of the predicates is still prohibitive.

4.3 Basic Merger

The *Merger* takes as input a list of predicates ranked by an internal score, merges subsets of the predicates, and returns the resulting list. Two predicates are merged by computing the minimum bounding box of the continuous attributes and the union of the values for each discrete attribute. The basic implementation repeatedly expands the existing predicates in decreasing order of their scores. Each predicate is expanded by greedily merging it with adjacent predicates until the resulting influence does not increase.

This implementation suffers from multiple performance-related issues if the aggregate is treated as a black-box. Each iteration calls the *Scorer* on the merged result of every pair of adjacent predicates, and may successfully merge only one pair of predicates. Section 6.3 explores optimizations that address these issues.

The next section will describe several aggregate operator properties that enable more efficient algorithm implementations.

5. AGGREGATE PROPERTIES

To compute results in a managable time, algorithms need to efficiently estimate a predicate's influence, and prune the space of predicates. These optimizations depend on the aggregate operator's properties. This subsection describes several properties that Scorpion exploits to develop more efficient algorithms. Aggregate developers only need to implement these properties once, and they are transparent to the end-user.

5.1 Incrementally Removable

The *Scorer* is extensively called from all of our algorithms, so reducing its cost is imperative. Rather than recomputing the aggregate function on the input dataset, an *incrementally removable* aggregate is able to directly evaluate p's influence from its tuples.

In general, a computation is *incrementally removable* if the updated result of removing a subset, s, from the inputs, D, can be computed by only reading s. For example, SUM is incrementally removable because $SUM(D-s) = SUM(D) - SUM(s)$, and the $SUM(D)$ component can be cached. In fact, computing influence of an aggregate is incrementally removable as long as the aggregate itself is incrementally removable.

Formally, an aggregate function, $\mathcal{F}$, is incrementally removable if it can be decomposed into functions $state$, $update$, $remove$ and $recover$, such that:

$$\begin{aligned} state(D) &\rightarrow m_D \\ update(m_{S_1}, \cdots, m_{S_n}) &\rightarrow m_{\cup_{i\in[1,n]}S_i} \\ remove(m_D, m_{S_1}) &\rightarrow m_{D-S_1} \\ \mathcal{F}(D) &= recover(m_D) \end{aligned}$$

Where D is the original dataset and $S_1 \cdots S_n$ are non-overlapping subsets of D to remove. $state$ computes a constant sized tuple that summarizes the aggregation operation, $update$ combines n tuples into one, $remove$ computes the tuple of removing S_1 from D, and $recover$ recomputes the aggregate result from the tuple. The *Scorer* takes advantage of this property to compute and cache $state(D)$ once. A predicate's influence is computed by removing the predicate's tuple from D's tuple, and calling $recover$ on the result. Section 6.3 describes cases where the *Merger* can use tuples to avoid calling the *Scorer* altogether.

This definition is related to the concept of distributive and algebraic functions in OLAP cubes [6], which are functions where a sub-aggregate can be stored in a constant bound, and composed to compute the complete aggregate. In contrast *incrementally removable* functions only need a constant bound to store the complete aggregate so that sub-aggregates can be removed. Although similar, not all distributive or algebraic are incrementally removable. For example, it is not in general possible to re-compute MAX after removing an arbitrary subset of inputs without knowledge of the full dataset. Similarly, MIN, $MEDIAN$ and $MODE$ are not incrementally-removable. In general, $COUNT$ and SUM based arithmetic expressions, such as AVG, $STDDEV$, and $VARIANCE$ are incrementally removable.

An operator developer can define an incrementally removable operator by implementing the procedures $state$, $update$, $remove$ and $recover$, which Scorpion uses to efficiently compute the influence of a predicate. For example, AVG is augmented as:

$$\begin{aligned} AVG.state(D) &= [SUM(D), |D|] \\ AVG.update(m_1, \cdots, m_n) &= [\textstyle\sum_{i\in[1,n]} m_i[0], \sum_{i\in[i,n]} m_i[1]] \\ AVG.remove(m_1, m_2) &= [m_1[0] - m_2[0], m_1[1] - m_2[0]] \\ AVG.recover(m) &= m[0]/m[1] \end{aligned}$$

5.2 Independent

The IP problem is non-trivial because combinations of input tuples can influence a user-defined aggregate's result in arbitrary ways. The *independence* property allows Scorpion to assume that the input tuples influence the aggregate result independently.

Let $t_1 \leq \cdots \leq t_n$ such that $\forall_{i\in[1,n-1]} inf_{\mathcal{F}}(o, t_i) \leq inf_{\mathcal{F}}(o, t_{i+1})$ be an ordering of tuples in input group, g_o, by their influence on the result o, where $\mathcal{F}$ is an aggregate computation. Let T be a set of tuples, then $\mathcal{F}$ is independent if:

1. $t_a < t_b \rightarrow inf_{\mathcal{F}}(T \cup \{t_a\}) < inf_{\mathcal{F}}(T \cup \{t_b\})$ and
2. $t_a > t^* \rightarrow inf_{\mathcal{F}}(T \cup \{t_a\}) \geq inf_{\mathcal{F}}(T) | t^* = \arg\min_{t\in T} inf_{\mathcal{F}}(t)$

The first requirement states that the influence of a set of tuples strictly depends on the influences of the individual tuples without regard to the tuples in T (they do not interact with t_a or t_b). The second specifies that adding a tuple t_a more influential than $t^* \in T$ with minimum influence can't decrease the set's influence.

Together, these requirements point towards a greedy strategy to find the most influential set of tuples for independent aggregates. Assume that the user provided a single suspicious result and no hold-outs. The algorithm first sorts the tuples by influence and then incrementally adds the most influential tuple to the candidiate set until the influence of the set does not increase further. This algorithm is guaranteed to find the optimal set (though not necessarily the optimal predicate).

While this sounds promising, the above requirements are difficult to guarantee because they depend on internal details such as the cardinality of the predicate and the existance of hold-out results. We instead modify the requirements to depend on $\Delta_{\mathcal{F}}$, which only depends on the aggregate function, rather than $inf_{\mathcal{F}}$. The developer specifies that an operator is independent by setting the attribute, $is\_independent = True$. The *DT* partitioning algorithm described in Section 6.1 exploits this property for aggregates such as *AVG* and *STDDEV*.

5.3 Anti-monotonic Influence

The anti-monotonic property is used to prune the search space of predicates. In general, a property is anti-monotone if, whenever a set of tuples s violates the property, so does any subset of s. In our context, an operator is anti-monotonic if for a given input group, g_o, the amount that a predicate, p, influences the aggregate result, $inf(o, p)$, is greater than or equal to the influence of any predicate contained within p:

$$p' \prec p \leftrightarrow inf(p') \leq inf(p)$$

In other words, if p is non-influential, then none of the predicates contained in p can be influential, and p can be pruned. For example, if D is a set of non-negative values, then $SUM(D) > SUM(s) \forall_{s\subseteq D}$. Note that the property only holds if the data satisfies a non-negativity constraint.

Similar to the *independence* property, determining anti-monotonicity at the influence level is non-trivial. Thus, developers only specify if Δ_{agg} obeys this property by defining a boolean function $check(D)$, that returns $True$ if D satisfies any required constraints, and $False$ otherwise. The function would be defined for anti-monotonic statistical functions as:

$$COUNT.check(D) = True$$
$$MAX.check(D) = True$$
$$SUM.check(D) = |\{d \in D | d < 0\}| == 0$$

6. IP ALGORITHMS

While the general IP problem is exponential, the properties presented in the previous section enable several more efficient partitioning and merging algorithms. In this section, we describe a top-down partitioning algorithm that takes advantage of independent operators and a bottom-up algorithm for independent, anti-monotonic aggregates. We then describe optimizations, including one to the basic *Merger* for certain independent aggregates.

6.1 Decision Tree (DT) Partitioner

DT is a top-down partitioning algorithm for independent aggregates. It is based on the intuition that the Δ function of independent operators cannot decrease when tuples with similar influence are combined together. *DT* generates predicates where the tuples of an input group within a predicate have similar influence. The *Merger* then greedily merges adjacent predicates with similar influence to produce the final predicates.

DT recursively splits the attribute space of an input group to create a set of predicates. Because outlier groups are different than hold-out groups, we partition these groups separately, resulting in a set of outlier predicates and hold-out predicates. These are combined into a set of predicates that differentiates ones that only influence outlier results from those that also influence hold-out results. We first describe the partitioning algorithm for a single input group, then for a set of outlier input groups (or hold-out input groups), and finally how to combine outlier and hold-out partitionings.

6.1.1 Recursive Partitioning

The recursive partitioner takes an input group, aggregate, and error vector (for outliers) as input, and returns a partitioning[4] such that the variance of the influence of individual tuples within a partition is less than a small multiplicative bound, relative to the average tuple's influence in the partition. Our algorithm is based on regression tree algorithms, so we first explain a typical regression tree algorithm before describing our differences.

Regression trees [2] are the continuous counterpart to decision trees, used to predict a continuous attribute rather than a categorical attribute. Initially, the tree begins with all data in a single cell or *node*. The algorithm fits a constant or linear formula to the tuples in the node, and computes an error metric over the tuple values (e.g., standard error or sum error). If the error metric or number of tuples in the node are below their respective thresholds, then the algorithm stops. Otherwise, the algorithm computes the best (attribute, value) pair to bisect the node, such that the resulting *child nodes* minimize the error metric, and recursively calls the algorithm on the children.

Our approach re-uses the regression tree framework to minimize the distribution of influence values within a given partition. The key insight is that it is more important that partitions containing influential tuples be accurate than non-influential partitions, thus the error metric threshold can be relaxed for partitions that don't contain any influential tuples. Its value is based on the maximum influence in a partition, inf_{max}, and the upper, inf_u, and lower, inf_l, bounds of the influence values in the dataset. The threshold can be computed via any function that decreases from a maximum to a minimum threshold value as inf_{max} approaches inf_u. Scorpion computes the threshold as:

$$threshold = \omega * (inf_u - inf_l)$$
$$\omega = min(\tau_{min} + s * (inf_u - inf_{max}), \tau_{max})$$
$$s = \frac{\tau_{min} - \tau_{max}}{(1-p) * inf_u - p * inf_l}$$

where ω is the multiplicative error as depicted in Figure 4, s is the slope of the downward curve, $p = 0.5$ is the inflection point when the threshold starts to decrease, and τ_{max} and τ_{min} are the maximum and minimum threshold values.

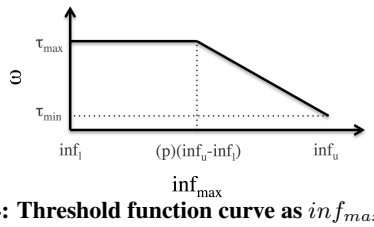

Figure 4: Threshold function curve as inf_{max} varies

6.1.2 Recursive Partitioning with Sampling

The previous algorithm still needs to compute the influence on all of the input tuples. To reduce this cost, we exploit the observation that the influential tuples should be clustered together (since Scorpion searches for predicates), and sample the data in order to avoid processing all non-influential tuples. The algorithm uses an additional parameter, ϵ, that represents the maximum number of influential tuples in a cluster as a percentage of the dataset. The system initially estimates a sampling rate, $samp\_rate$, such that a sample from D of size $samp\_rate * |D|$ will contain high influence tuples with high probability ($\geq 95\%$):

$$sample\_rate = min(\{sr | sr \in [0,1] \wedge 1 - (1-\epsilon)^{sr*|D|} \geq 0.95\})$$

Scorpion initially uniformly samples the data, however after computing the influences of the tuples in the sample, there is information about the distribution of influences. We use this when splitting a partition to determine the sampling rate for the sub-partitions. In particular, we use stratified sampling, weighed on the total relative influences of the samples that fall into each sub-partition.

To illustrate, let D be partitioned by the predicate p into $D_1 = p(D)$ and $D_2 = \neg p(D)$, and $S \subset D$ be the sample with sampling rate $samp\_rate$. We use the sample to estimate D_1's (and similarly D_2's) total influence:

$$inf_{D_1} = \sum_{t \in p(S)} inf(t)$$

The sampling rate for D_1 (and similarly D_2) is computed as:

$$samp\_rate_{D_1} = \frac{inf_{D_1}}{inf_{D_1} + inf_{D_2}} * \frac{|S|}{|D_1|}$$

6.1.3 Parallel Partitioning and Synchronization

DT separately partitions outlier from hold-out input groups to avoid the complexity of computing the combined influence. It is tempting to compute the union of the input groups and execute the above recursive partitioner on the resulting set, however, it can result in over-partitioning. For example, consider α_2 and α_3 from Table 2. The outlier temperature readings (T6 and T9) are correlated with low voltage. If g_{α_2} and g_{α_3} are combined, then the error

[4] Partitions and predicates are interchangeable, however the term partition is more natural when discussing space partitioning algorithms.

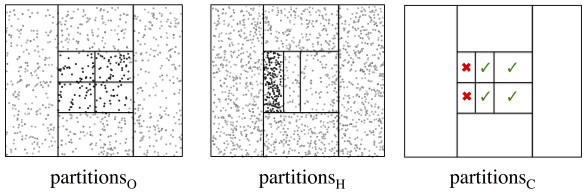

Figure 5: Combined partitions of two simple outlier and hold-out partitionings

metric of the predicate $voltage < 2.4$ would still have high variance, and be falsely split further. In the worst case, the partitioner will create single-tuple partitions.

Instead, we execute a separate instance of the partitioning algorithm on each input group. Before picking a split attribute for a partition, the algorithms combine the error metrics computed for each candidate attribute to select a single best split. This ensures that the algorithms output the same partitioning. The error metrics for attribute $attr$ are combined as $metric_{attr} = max(metric^i_{attr} | i \in [0, |R|])$, where $metric^i_{attr}$ is the error metric of attribute a in the i'th input group's algorithm.

6.1.4 Combining Partitionings

The recursive partitioning step generates an outlier partitioning, $partitions_O$, and hold-out partitioning, $partitions_H$, for their respective input groups. The final step is to combine them together into a single partitioning, $partitions_C$. The goal is to distinguish partitions that influence hold-out results from those that only influence outlier results. We do this by splitting partitions in $partitions_O$ along their intersections with partitions in $partitions_H$.

For example, $partitions_H$ in Figure 5 contains a partition that overlaps with two of the influential partitions in $partitions_O$. The splitting process distinguishes partitions that influence hold-out results (contains a red 'X') from those that only influence outlier results (contains a greed check mark).

6.2 Bottom-Up (MC) Partitioner

The *MC* algorithm is a bottom-up approach for independent, anti-monotonic aggregates, such as $COUNT$ and SUM. It can be much more efficient than *DT* for these aggregates. The idea is to first search for influential single-attribute predicates, then intersect them to construct multi-attribute predicates. Our technique is similar to algorithms used for subspace clustering [1], so we will first sketch a classic subspace clustering algorithm, and then describe our modifications. The output is then sent to the *Merger*.

The subspace clustering problem searches for all subspaces (hyper-rectangles) that are denser than a user defined threshold. The original algorithm, CLIQUE [1], and subsequent improvements, employs a bottom-up iterative approach that initially splits each continuous attribute into fixed size units, and every discrete attribute by the number of distinct attribute values. Each iteration computes the intersection of all units kept from the previous iteration whose dimensionality differ by exactly one attribute. Thus, the dimensionality of the units increase by one after each iteration. Non-dense units are pruned, and the remaining units are kept for the next iteration. The algorithm continues until no dense units are left. Finally, adjacent units with the same dimensionality are merged. The pruning step is possible because density (i.e. $COUNT$) is anti-monotonic because non-dense regions cannot contain dense sub-regions.

The intuition is to start with coarse-grained predicates (single dimensional), and improve the influence by adding additional dimensions that refine the predicates. We have two major modifications to the subspace clustering algorithm. First, we merge adjacent

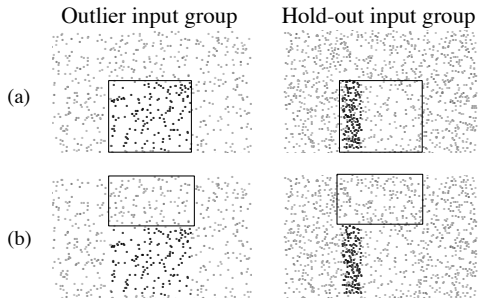

Figure 6: A predicate is not influential if it (a) influences a hold-out result or (b) doesn't influence outlier result.

units after each iteration to find the most influential predicate. If the merged predicate is not more influential than the optimal predicate so far, then the algorithm terminates.

Second, we modify the pruning procedure to account for two ways in which the influence metric is not anti-monotonic. The first case is when the user specifies a hold-out set. Consider the problem with a single outlier result, o, and a single hold-out result, h (Figure 6). A predicate, p, may be non-influential because it also influences a hold-out result (Figure 6.a), or because it doesn't influence the outlier result (Figure 6.b). In the former case, there may exist a predicate, $p' \prec_{g_o \cup g_h} p$ that only influences the outlier results. Pruning p would mistakenly also prune p'. In the latter case, p can be safely pruned. We distinguish these cases by pruning p based on its influence over only the outlier results, which is a conservative estimate of p's true influence.

```
1:  function MC(O, H, V)
2:      predicates ← Null
3:      best ← Null
4:      while |predicates| > 0 do
5:          if predicates = Null then
6:              predicates ← initialize_predicates(O, H)
7:          else
8:              predicates ← intersect(predicates)
9:          best ← arg max_{p∈merged} inf(p)
10:         predicates ← prune(predicates, O, V, best)
11:         merged ← Merger(predicates)
12:         merged ← {p|p ∈ merged ∧ inf(p) > inf(best)}
13:         if merged.length = 0 then
14:             break
15:         predicates ← {p|∃_{p_m∈merged} p ≺_D p_m}
16:         best ← arg max_{p∈merged} inf(p)
17:     return best
18:
19: function PRUNE(predicates, O, V, best)
20:     ret = {p ∈ predicates|inf(O, ∅, p, V) < inf(best)}
21:     ret = {p ∈ ret| arg max_{t*∈p(O)} inf(t*) < inf(best)}
22:     return ret
```

The second case is because anti-monotonicity is defined for $\Delta(p)$, however influence is proportional to $\frac{\Delta(p)}{|p|}$, which is not anti-monotonic. For example, consider three tuples with influences, $\{1, 50, 100\}$ and the operator SUM. The set's influence is $\frac{(1+50+100)}{3} = 50.3$, whereas the subset $\{50, 100\}$ has a higher influence of 75. It turns out that the anti-monotonicity property holds if, for a set of tuples, s, the tuple with the maximum influence, $t^* = \arg\max_{t \in s} inf(t)$ is less than the influence of s: $inf(t^*) < inf(s)$.

The *MC* algorithm is shown above. The first iteration of the $WHILE$ loop initializes $predicates$ to the set of single attribute predicates and subsequent iterations intersect all pairs in $predicates$ (Lines 5-8). The best predicate so far, $best$, is updated, and then used to prune $predicates$ (Lines 9,10). The resulting predicates are merged, and filtered for ones that are more influential than $best$

(Lines 11-12). If none of the merged predicates are more influential than $best$, then the algorithm terminates. Otherwise $predicates$ and $best$ are updated, and the next iteration proceeds.

The pruning step first removes predicates whose influence, ignoring the hold-out sets, is less than the influence of $best$. It then removes those that don't contain a tuple whose individual influence is greater than $best$'s influence.

6.3 Merger Optimizations

We now present optimizations to the *Merger* when its inputs are generated by the *DT* algorithm. As described in Section 4.3, the *Merger* scans its list of predicates and expands each one by repeatedly merging it with its adjacent predicates. The first optimization reduces the number of predicates that need to be merged by only expanding the predicates whose influences are within the top quartile. This is based on the intuition that the final predicate is most likely to influence predicates in the top quartile, so it is inefficient to expand less influential predicates.

The second optimization seeks to completely avoid calling the *Scorer* when the operator is also incrementally removable (e.g., AVG, $STDDEV$). Although the incrementally removable property already avoids recomputing the aggregate over the entire dataset, computing a predicate's influence still requires executing the predicate on the underlying dataset. Doing this repeatedly for every pair of merged predicates is slow.

Recall that *DT* generates partitions where the tuples in a partition have similar influence. We modify *DT* to additionally record each partition's cardinality, and the tuple whose influence is closest to the mean influence of the partition. The *Merger* can then use the aggregate's $state, update, remove$ and $recover$ functions to directly approximate the influence of a partition from the cached tuple. Concretely, suppose a partition, p, has cardinality N and cached tuple, t. Let $m_t = state(t)$ and $m_D = state(D)$ be the states of $\{t\}$ and the dataset, then

$$inf(p) \approx recover(remove(m_D, update(m_t, \cdots, m_t)))$$

where $update$ combines N copies of m_t. In other words, p's influence can be approximated by combining N copies of m_t, removing them from m_D, and calling $recover$. Now consider merging partitions p_1 and p_2 into p^* as shown in Figure 7 and approximating its influence. This scenario is typically difficult because its not clear how the tuples in p_3 and $p_1 \cap p_2$ affect p^*'s influence. Similar to replicating the cached tuple multiple times to approximate a single partition, we estimate the number of cached tuples that p_1,p_2, and p_3 contribute. We assume that tuples are distributed uniformly within the partitions. Let V_p and N_p be the volume and cardinality of partition p and let p_{12} be a shorthand for $p_1 \cap p_2$. Then the number of cached tuples from each partition, n_p is computed as follows:

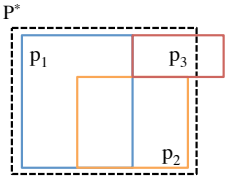

Figure 7: Merging partitions p_1 and p_2

$$n_{p_1} = N_{p_1} \frac{V_{p_1} - 0.5V_{p_{12}}}{V_{p^*}}$$

$$n_{p_2} = N_{p_2} \frac{V_{p_2} - 0.5V_{p_{12}}}{V_{p^*}}$$

$$n_{p_3} = N_{p_3} \frac{V_{p_3 \cap p^*}}{V_{p^*}}$$

The *Merger* approximates a partition's influence from the input partitions by estimating the number of cached-tuples that each input partition contributes. Thus, the cost only depends on the number of intersecting partitions, rather than the size of the dataset.

6.4 Dimensionality Reduction

An additional optimization that is orthogonal to the particular algorithm that is used is to reduce the number of attributes that Scorpion needs to consider. This can be done by applying filter based feature selection techniques [13] to the dataset. These techniques identify non-informative features by computing correlation or mutual information scores between pairs of attributes. Alternatively, users can manually order the attributes by importance. This often makes sense because the user can distinguish informative or actionable attributes (e.g., sensor id) from non-informative ones (e.g., epoch, which is redundant with the timestamp). Scorpion currently relies on users to specify attributes that can be ignored, and defers incorporating automatic attribute selection to future work.

7. AN ADDITIONAL KNOB

We have not yet addressed what to do if the user specifies that an outlier result is "too high". Specifically, how low should the updated result go? Throughout this paper, we have defined the basic definition of influence as $inf_{agg}(o,p) = \frac{\Delta o}{\Delta g_o}$ where the amount that the outlier changes is penalized by the number of tuples that are used. However the user may not care about the number of tuples, or be willing to accept a predicate that matches more tuples if it increases the influence. We modify the basic definition of influence as follows:

$$inf_{agg}(o,p,c) = \frac{\Delta o}{(\Delta g_o)^c}$$

The exponent, $c \geq 0$, is a user-controlled parameter that trades off the importance of keeping the size of s small and maximizing the change in the aggregate result. In effect, when a user specifies that an outlier result is too high, c controls how aggressively Scorpion should reduce the result. For example, when $c = 0$, Scorpion will reduce the aggregate result without regard to the number of tuples that are used, producing predicates that select many tuples. Increasing c places more emphasis on finding a smaller set of tuples, producing much more selective predicates.

Although c is a crucial parameter, none of the algorithms need to be modified to support its addition. In fact, we will show in the experiments that MC benefits from low c values because it decreases the denominator in the basic influence definition, and thus increases the pruning threshold in the algorithm. We study the effects of c extensively in the experiments.

8. EXPERIMENTS

The goal of these experiments is to gain an understanding of how the *NAIVE*, *DT* and *MC* algorithms compare in terms of performance and answer quality. Furthermore, we want to understand how the c parameter impacts the types of predicates that the algorithms generate. We first use a synthetic dataset with varying dimensionality and task difficulty to analyze the algorithms, then, due to space constraints, anecdotally comment on the result qualities on 4 and 12 dimensional real-world datasets.

8.1 Datasets

This subsection describes each dataset in terms of the schemas, attributes, user queries, and properties of the outlier tuples.

SYNTH: The synthetic dataset is used to generate ground truth data to compare our various algorithms. The SQL query contains an independent anti-monotonic aggregate and is of the form:

$$\text{SELECT SUM}(\text{A}_\text{v})\ \text{FROM synthetic GROUP BY A}_\text{d}$$

The data consists of a single group-by attribute, A_d, one value attribute, A_v, that is used to compute the aggregate result, and n dimension attributes, $A_1, \cdots, A_n$ that are used to generate the explanatory predicates. The value and dimension attributes have a domain of $[0, 100]$. We generate 10 distinct A_d values (to create 10 groups), and each group contains 2,000 tuples randomly distributed in the n dimensions. The A_v values are drawn from one of three gaussian distributions, depending on if the tuple is a normal or outlier tuple, and the type of outlier. Normal tuples are drawn from $\mathcal{N}(10, 10)$. To illustrate the effects of the c parameter we generate high-valued outliers, drawn from $\mathcal{N}(\mu, 10)$, and medium valued outliers, drawn from $\mathcal{N}(\frac{\mu+10}{2}, 10)$. $\mu > 10$ is a parameter to vary the difficulty of distinguishing normal and outlier tuples. The problem is harder the closer μ is to 10. Half of the groups (the hold-out groups) exclusively sample from the normal distribution, while the rest (the outlier groups) sample from all three distributions.

We generate the outlier groups by creating two random n dimensional hyper-cubes over the n attributes where one is nested inside the other. The *outer cube* contains 25% of the tuples in the group, and the *inner cube* contains 25% of the tuples in the outer cube. The A_v values of tuples in the inner cube are high valued outliers, while those of the outer cube are medium valued. The rest of the tuples in the group are normal. For example, Figure 8 visualizes a synthetic 2D dataset with $\mu = 90$, the outer cube as $A_1 \in [20, 80], A_2 \in [20, 80]$ and the inner cube as $A_1 \in [40, 60], A_2 \in [40, 60]$. The top is a graph of the aggregate results the that user would see, and bottom shows input tuples of one outlier result and one hold-out result.

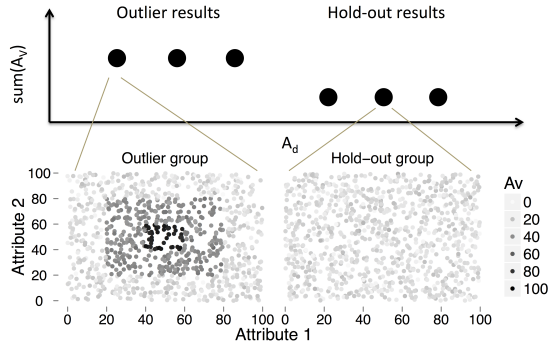

Figure 8: Visualization of outlier and hold-out results and tuples in their input groups from a 2-D synthetic dataset. The colors represent normal tuples (light grey), medium valued outliers (grey), and high valued outliers (black).

We flag the 5 outlier aggregate results, and use the other 5 as hold-outs. We also vary the dimensionality from 2 to 4, and the difficulty between *Easy* ($\mu = 80$) and *Hard* ($\mu = 30$), e.g., *SYNTH-2D-Easy* generates a 2-dimensional dataset where $\mu = 80$.

INTEL: The Intel dataset contains 2.3 million rows, and 6 attributes. Four of the attributes, $sensorid$, $humidity$, $light$, and $voltage$ are used to construct explanations. All of the attributes are continuous, except for $sensorid$, which contains ids of the 61 sensors.

We use two queries for this experiment, both related to the impact of sensor failures on the standard deviation of the temperature. The following is the general query template, and contains an independent aggregate:

$$\text{SELECT truncate}('\text{hour}',\ \text{time})\ \text{as hour, STDDEV(temp)}$$
$$\text{FROM readings}$$
$$\text{WHERE } STARTDATE \leq \text{time} \leq ENDDATE \text{ GROUP BY hour}$$

The first query occurs when a single sensor ($sensorid = 15$) starts dying and generating temperatures above $100^o c$. The user selects 20 outliers and 13 hold-out results, and specifies that the outliers are too high.

The second query is when a sensor starts to lose battery power, indicated by low voltage readings, which causes above $100^o c$ temperature readings. The user selects 138 outliers and 21 hold-out results, and indicates that the outliers are too high.

EXPENSE: The expenses dataset [5] contains all campaign expenses between January 2011 and July 2012 from the 2012 US Presidential Election. The dataset contains 116448 rows and 14 attributes (e.g., recipient name, dollar amount, state, zip code, organization type), of which 12 are used to create explanations. The attributes are nearly all discrete, and vary in cardinality from 2 to 18 thousand (recipient names). Two of the attributes contain 100 distinct values, and another contains 2000.

The SQL query uses an independent, anti-monotonic aggregate and sums the total expenses per day in the Obama campaign. It shows that although the typical spending is around $5,000 per day, campaign spent up to $13 million per day on media-related purchases (TV ads) in June.

$$\text{SELECT sum(disb\_amt)}$$
$$\text{FROM expenses WHERE candidate} = '\text{Obama}'$$
$$\text{GROUP BY date}$$

We flag 7 outlier days where the expenditures are over $10M, and 27 hold-out results from typical days.

8.2 Experimental Setup and Methodology

Our experiments compare Scorpion using the three partitioning algorithms along metrics of precision, recall, F-score and runtime. We compute precision and recall of a predicate, p, by comparing the set of tuples in $p(g_O)$ to a ground truth set. The F-score is defined as the harmonic mean of the precision and recall:

$$F = 2 \times \frac{precision \times recall}{precision + recall}$$

The *NAIVE* algorithm described in Section 4.2 is clearly exponential and is unacceptably slow for any non-trivial dataset. We modified the exhaustive algorithm to generate predicates in order of increasing complexity, where complexity is terms of the number and size of values in a discrete clause, and the number of clauses in the predicate. The modified algorithm uses two outer loops that increases the maximum allowed complexity of the discrete clauses and the maximum number of attributes in a predicate, respectively, and an inner loop that iterates through all combinations of attributes and their clauses. When the algorithm has executed for a user specified period of time, it terminates and returns the most influential predicate generated so far. In our experiments, we ran the exhaustive algorithm for up to 40 minutes, and also logged the best predicate at every 10 second interval.

The current Scorpion prototype is implemented in Python 2.7 as part of an end-to-end data exploration tool. The experiments are run single threaded on a Macbook Pro (OS-X Lion, 8GB RAM). The *Naive* and *MC* partitioner algorithms were configured to split each continuous attribute's domain into 15 equi-sized ranges.

[5] http://www.fec.gov/disclosurep/PDownload.do

8.3 Synthetic Dataset

Our first set of experiments use the 2D synthetic datasets to highlight how the c parameter impacts the quality of the optimal predicate. We execute the *NAIVE* algorithm until completion and show how the predicates and accuracy statistics vary with different c values. The second set of experiments compare the *DT*, *MC* and *NAIVE* algorithms by varying the dimensionality of the dataset and the c parameter. The final experiment introduces a caching based optimization for the *DT* algorithm and the *Merger*.

8.3.1 Naive Algorithm

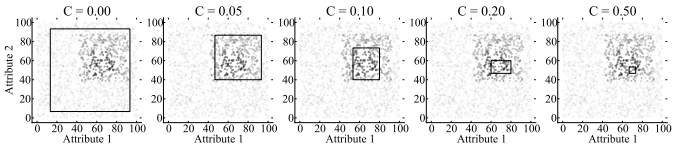

Figure 9: Optimal NAIVE predicates for SYNTH-2D-Hard

Figure 9 plots the optimal predicate that $Naive$ finds for different c values on the SYNTH-2D-Hard dataset. When $c = 0$, the predicate encloses all of the outer cube, at the expense of including many normal points. When $c = 0.05$, the predicate contains most of the outer cube, but avoids regions that also contain normal points. Increasing c further reduces the predicate and exclusively selects portions of the inner cube.

It is important to note that all of these predicates are correct and influence the outlier results to a different degree because of the c parameter. This highlights the fact that a single ground truth doesn't exist. For this reason, we simply use the tuples in the inner and outer cubes of the synthetic datasets as surrogates for ground truth.

Figure 10 plots the accuracy statistics as c increases. Each column of figures plots the results of a dataset, and each curve uses the outer or inner cube as the ground truth when computing the accuracy statistics. Note that for each dataset, the points for the same c value represent the same predicate. As expected, the F-score of the outer curve peaks at a lower c value than the inner curve. This is because the precision of the outer curve quickly approaches 1.0, and further increasing c simply reduces the recall. In contrast, the recall of the inner curve is maximized at lower values of c and reduces at a slower pace. The precision statistics of the inner curve on the Easy dataset increases at a slower rate because the value of the outliers are much higher than the normal tuples, which increases the predicate's Δ values.

Figure 11 depicts the amount of time it takes for Naive to converge when executing on SYNTH-2D-Hard. The left column computes the accuracy statistics using the inner cube as ground truth, and the right column uses the outer cube. Each point plots the accuracy score of the most influential predicate so far, and each curve is for a different c value. *NAIVE* tends to converge faster when c is close to zero, because the optimal predicate involves fewer attributes. The curves are not monotonically increasing because the the optimal predicate as computed by influence does not perfectly correlate with the ground truth that we selected.

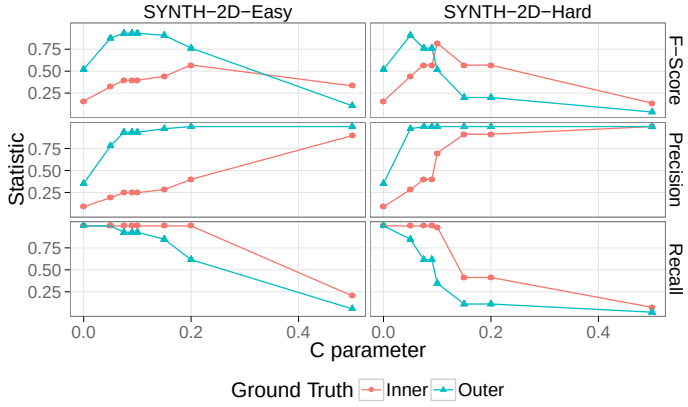

Figure 10: Accuracy statistics of NAIVE as c varies using two sets of ground truth data.

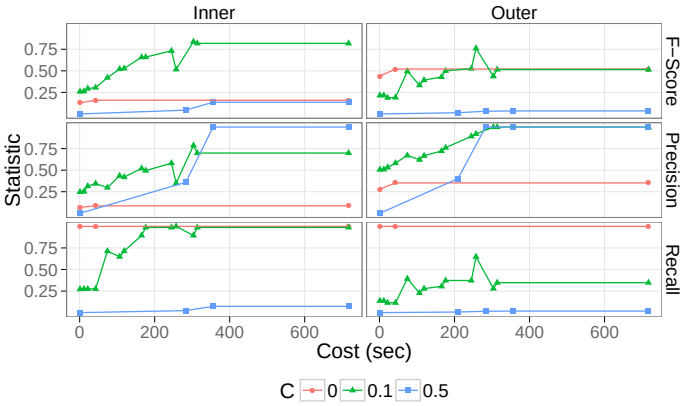

Figure 11: Accuracy statistics as execution time increases for NAIVE on SYNTH-2D-Hard

Takeaway: Although the F-score is a good proxy for result quality, it can be artificially low depending on the value of c. Although NAIVE converges (relatively) quickly when c is very low, it is very slow at high c values.

8.3.2 Comparing Algorithms

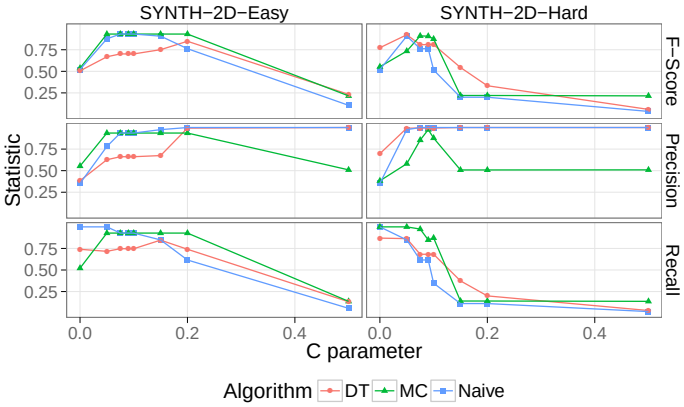

Figure 12: Accuracy measures as c varies

The following experiments compare the accuracy and runtime of the *DT*, *MC* and *NAIVE* algorithms. Figure 12 varies the c parameter and computes the accuracy statistics using the outer cube as the ground truth. Both *DT* and *MC* generate results comparable with those from the *NAIVE* algorithm. In particular, the maximum F-scores are similar.

Figure 13 compares the F-scores of the algorithms as the dimensionality varies from 2 to 4. Each row and column of plots corresponds to the dimensionality and difficulty of the dataset, respectively. As the dimensionality increases, *DT* and *MC* remain competitive with *NAIVE*. In fact, in some cases *DT* produces better results than *NAIVE*. Partly because because *NAIVE* splits each attribute into a pre-defined number of intervals, whereas *DT* can split the predicates into any granularity, and partly because *NAIVE*

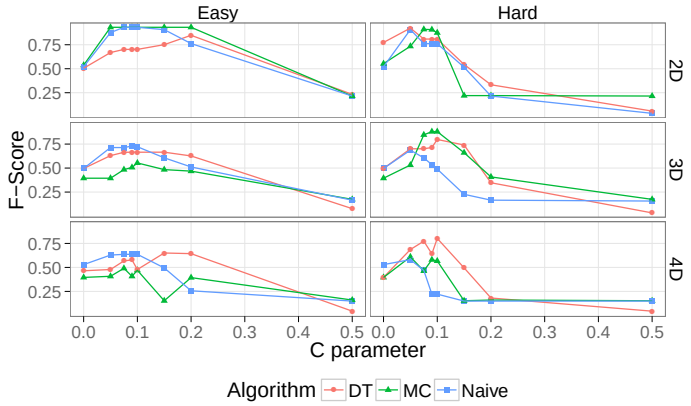

Figure 13: F-score as dimensionality of dataset increases

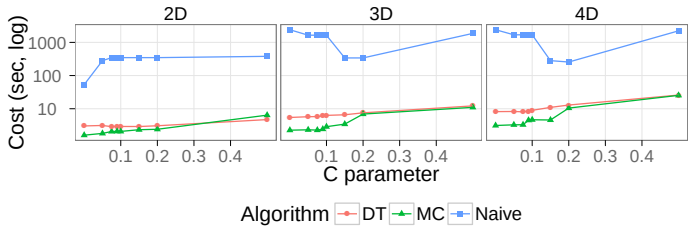

Figure 14: Cost as dimensionality of dataset increases

doesn't terminate within the 40 minutes at higher dimensions – running it to completion would generate the optimal predicate.

Figure 14 compares the algorithm runtimes while varying the dimensionality of the Easy synthetic datasets. The *NAIVE* curve reports the earliest time that *NAIVE* converges on the predicate returned when the algorithm terminates. We can see that *DT* and *MC* are up to two orders of magnitude faster than Naive. We can also see how *MC*'s runtime increases as c increases because there are less opportunities to prune candidate predicates.

Figure 15 uses the Easy datasets and varies the number of tuples per group from 500 (5k total tuples) to 10k (100k total tuples) for a fixed $c = 0.1$. The runtime is linear with the dataset size, but the slope increases super-linearly with the dimensionality because the number of possible splits and and merges increases similarly. We found that *DT* spends significant time splitting non-influential partitions because the standard deviation of the tuple samples are too high. When we re-ran the experiment by reducing the variability by drawing normal tuples from $\mathcal{N}(10, 0)$ reduces the runtime by up to 2×. We leave more advanced optimization techniques, e.g., early pruning, parallelism to future work.

Takeaway: DT and MC generate results competitive with the exhaustive NAIVE algorithm and reduces runtime costs by up to 150×. Algorithm performance relies on data properties, and scales exponentially with the dimensionality in the worst case. DT's results may have higher F-scores than NAIVE because it can progressively refine the predicate granularity.

8.3.3 Caching Optimization

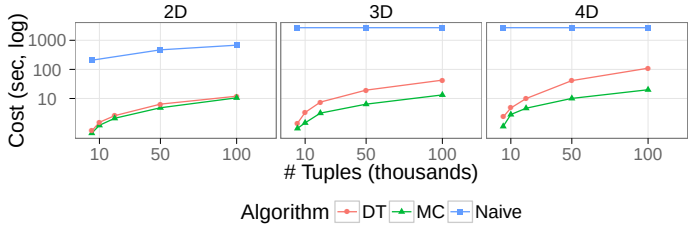

Figure 15: Cost as size of Easy dataset increases (c=0.1)

The previous experiments showed that the result predicates are sensitive to c, thus the user or system may want to try different values of c (e.g., via a slider in the UI or automatically). *DT* can cache and re-use its results because the partitioning algorithm is agnostic to the c parameter. Thus, the *DT* partitioner only needs to execute once for Scorpion queries that only change c.

The *Merger* can similarly cache its previous results because executes iteratively in a deterministic fashion – increasing the c parameter simply reduces the number of iterations that are executed. Thus Scorpion can initialize the merging process to the results of any prior execution with a higher c value. For example, if the user first ran a Scorpion query with $c = 1$, then those results can be re-used when the user reduces c to 0.5.

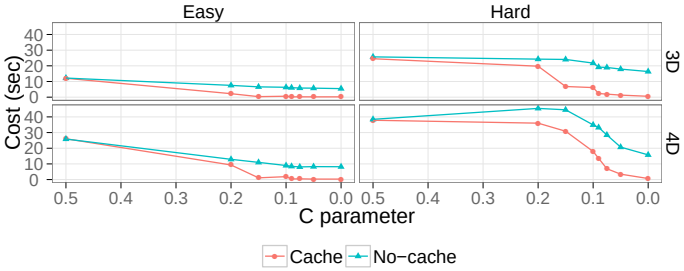

Figure 16: Cost with and without caching enabled

Figure 16 executes Scorpion using the *DT* partitioner on the synthetic datasets. We execute on each dataset with decreasing values of c (from 0.5 to 0), and cache the results so that each execution can benefit from the previous one. Each sub-figure compares Scorpion with and without caching. It is most beneficial to cache *Merger* results at lower c values because more predicates are merged so there are less predicates to consider merging. When c is high, most predicates are not expanded, so the cache doesn't reduce the amount of work that needs to be done.

Takeaway: Caching DT and Merger results for low c values reduces execution cost by up to 25×.

8.4 Real-World Datasets

To understand how Scorpion performs on real-world datasets, we applied Scorpion to the INTEL and EXPENSES workloads. Since there is no ground truth, we present the predicates that are generated and comment on the predicate quality with respect to our expectations and further analyses. The algorithms all completed within a few seconds, so we focus on result quality rather than runtime. In each of the workloads, we vary c from 1 to 0, and record the resulting predicates.

For the first INTEL workload, the outliers are generated by Sensor 15, so Scorpion consistently returns the predicate $\text{sensorid} = 15$. However, when c approaches 1, Scorpion generates the predicate, $\text{light} \in [0, 923]$ & $\text{voltage} \in [2.307, 2.33]$ & $\text{sensorid} = 15$. It turns out that although Sensor 15 generates all of the high temperature readings, the temperatures vary between are $20^o c$ higher when its voltage, and surprisingly, light readings are lower.

In the second INTEL workload, Sensor 18 generates the anomalous readings. Scorpion returned $\text{light} \in [283, 354]$ & $\text{sensorid} = 18$ when $c = 1$. Sensor 18's voltage is abnormally low, which causes it to generate high temperature readings ($90^o c - 122^o c$). The readings are particularly high ($122^o c$) when the light levels are between 283 and 354. Scorpion returns $\text{sensorid} = 18.0$ at lower c values.

In both workloads, Scorpion identified the problematic sensors and distingushed between extreme and normal outlier readings.

In the EXPENSES workload, we defined the ground truth as

all tuples where the expense was greater than \$1.5M. The aggregate was SUM and all of the expenses were positive so we executed the *MC* algorithm. For c values between 1 and 0.2, Scorpion generated the predicate $\text{recipient\_st} =' \text{DC}' \ \& \ \text{recipient\_nm} =' \text{GMMB INC.}' \ \& \ \text{file\_num} = 800316 \ \& \ \text{disb\_desc} =' \text{MEDIA BUY}'$ Although the F-score is 0.6 due to low recall, this predicate best describes Obama's highest expenses. The campaign submitted two "GMMB INC." related expense reports. The report with file_num = 800316 spend an average of \$2.7M. When c is reduced below 0.1, the file_num clause is removed, and the predicate matches all $\$1 + M$ expenditures for an average expenditure of \$2.6M.

9. RELATED WORK

Scorpion is most closely related to notions of influence introduced in the context of data provenance [10, 12, 7] that define how the probability of a result tuple is influenced by an input tuple. For example, Meliou et al. [11] define influence in the context of boolean expressions, where an input tuple's influence over a result tuple is relative to the minimum number of additional tuples (a *contingency set*) that must also be added to or removed from the database to toggle the result tuple's existence.

In the context of probabilistic provenance [12, 7], where every tuple is associated with a probability of existence, a tuple's influence is defined as the ratio of the change in the result's probability over the change in the input tuple's probability. Thus all of these works are able to rank individual tuples by influence, but do not consider constructing predicates for explanatory purposes.

Sunita el al. apply statistical approaches to similar applications that explore and explain values in an OLAP data cube. iDiff [15] uses an information-theoretic approach to generate summary tuples that explain why two subcubes' values differ (e.g., higher or lower). Their cube exploration work [16] uses the user's previously seen subcubes during a drill-down session to estimate the expected values of further drill-down operations. The system recommends the subcube most differing from expectation, which can be viewed as an "explanation". RELAX [17] lets users specify subcube trends (e.g., drop in US sales from 1993 to 1994) and finds the coarsest context that exhibits the similar trend. Scorpion differs by explicitly using influence as the optimization metric, and supports additional information such as hold-out results and error vectors.

MRI [4] searches for a predicate over the user attributes (e.g., age, state, sex) that most explains average rating of a movie or product (e.g., IMDB ratings). Their work is optimized for the $AVG()$ operator and uses a randomized hill climbing algorithm to find the most influential cuboid in the rating's OLAP lattice.

PerfXplain [9] explains why some MapReduce [5] jobs ran faster or slower than others. The authors provide a query language that lets users easily label pairs of jobs as normal or outliers, and uses a decision tree to construct a predicate that best describes the outlier pairs. In contrast, Scorpion users label *aggregate* results as outliers or normal, and the system must infer the label of the input tuples.

Several projects approach the anomaly explanation problem from an HCI perspective. Profiler [8] is a data exploration tool that mines for anomalies and generates visualizations that exposes those outliers. Willett et al [18] use crowd workers to explain outlier trends in visualization in human understandable terms, and present seven strategies to improve worker-generated explanations.

10. CONCLUSION

As data becomes increasingly accessible, data analysis capabilities will shift from specialists into the hands of end-users. These users not only want to navigate and explore their data, but also probe and understand why outliers in their datasets exist. Scorpion helps users understand the origins of outliers in aggregate results computed over their data. In particular, we generate human-readable predicates to help *explain* outlier aggregate groups based on the attributes of tuples that contribute to the value of those groups, and introduced a notion influence for computing the effect of a tuple on an output value. Identifying tuples of maximum influence is difficult because the influence of a given tuple depends on the other tuples in the group, and so a naive algorithm requires iterating through all possible inputs to identify the set of tuples of maximum influence. We then described three aggregate operator properties that can be leveraged to develop efficient algorithms that construct influential predicates of nearly equal quality to the exhaustive algorithm using orders of magnitude less time. Our experiments on two real-world datasets show promising results, accurately finding predicates that "explain" the source of outliers in a sensor networking and campaign finance data set.

11. ACKNOWLEDGEMENTS

We want to thank Adam Marcus for valuable discussions, Alexandra Meliou for very helpful comments, and to the anonymous VLDB reviewers for the sound feedback.

12. REFERENCES

[1] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *DMKD*, pages 94–105, 1998.

[2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman & Hall, New York, NY, 1984.

[3] Y. Cui, J. Widom, and J. L. Viener. Tracing the lineage of view data in a warehousing environment. In *ACM Transactions on Database Systems*, 1997.

[4] M. Das, S. Amer-Yahia, G. Das, and C. Yu. Mri: Meaningful interpretations of collaborative ratings. In *PVLDB*, volume 4, 2011.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[6] J. Gray, A. Bosworth, A. Layman, D. Reichart, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. pages 152–159, 1996.

[7] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, pages 841–852, 2011.

[8] S. Kandel, R. Parikh, A. Paepcke, J. Hellerstein, and J. Heer. Profiler: Integrated statistical analysis and visualization for data quality assessment. In *Advanced Visual Interfaces*, 2012.

[9] N. Khoussainova, M. Balazinska, and D. Suciu. Perfxplain: debugging mapreduce job performance. *VLDB*, 5(7):598–609, Mar. 2012.

[10] A. Meliou, W. Gatterbauer, J. Y. Halpern, C. Koch, K. F. Moore, and D. Suciu. Causality in databases. In *IEEE Data Eng. Bull.*, volume 33, pages 59–67, 2010.

[11] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 505–516, New York, NY, USA, 2011. ACM.

[12] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proc. VLDB Endow.*, 1(1):797–808, Aug. 2008.

[13] Y. Saeys, I. n. Inza, and P. Larrañaga. A review of feature selection techniques in bioinformatics. *Bioinformatics*, 23(19):2507–2517, Sept. 2007.

[14] A. Saltelli. The critique of modelling and sensitivity analysis in the scientific discourse. an overview of good practices. *TAUC*, Oct. 2006.

[15] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, 1999.

[16] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. In *EDBT*, 1998.

[17] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, 2001.

[18] W. Willett, J. Heer, and M. Agrawala. Strategies for crowdsourcing social data analysis. In *SIGCHI*, pages 227–236, 2012.

[19] E. Wu, S. Madden, and M. Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *ICDE*, 2013.