On Detecting Cherry-picked Generalizations

Yin Lin
University of Michigan
irenelin@umich.edu

Brit Youngmann
Tel Aviv University
brity@cs.tau.ac.il

Yuval Moskovitch
University of Michigan
yuvalm@umich.edu

H. V. Jagadish
University of Michigan
jag@umich.edu

Tova Milo
Tel Aviv University
milo@cs.tau.ac.il

ABSTRACT

Generalizing from detailed data to statements in a broader context is often critical for users to make sense of large data sets. Correspondingly, poorly constructed generalizations might convey misleading information even if the statements are technically supported by the data. For example, a cherry-picked level of aggregation could obscure substantial sub-groups that oppose the generalization. We present a framework for detecting and explaining cherry-picked generalizations by refining aggregate queries. We present a scoring method to indicate the appropriateness of the generalizations. We design efficient algorithms for score computation. For providing a better understanding of the resulting score, we also formulate practical explanation tasks to disclose significant counterexamples and provide better alternatives to the statement. We conduct experiments using real-world data sets and examples to show the effectiveness of our proposed evaluation metric and the efficiency of our algorithmic framework.

PVLDB Reference Format:
Yin Lin, Brit Youngmann, Yuval Moskovitch, H. V. Jagadish, Tova Milo. On Detecting Cherry-picked Generalizations. PVLDB, 15(1): 59-71, 2022.
doi:10.14778/3485450.3485457

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/niceIrene/Cherry-picked-Generalizations.

1 INTRODUCTION

Given a large data set, we make sense of it through aggregate statements based on the data. We refer to these aggregate statements as *generalizations*. For example, given a data set of people with height and gender, we may arrive at a generalization that, on average, men are taller than women. Of course we know there are many exceptions – in pairwise individual comparisons, we will find many women taller than men. (We may also worry about the database treating gender as binary and static). Nevertheless, the generalization may still be a "reasonable" conclusion of the data.

The natural question then is to determine when a generalization is reasonable? So-called "hasty generalization" has been identified as a common mistake made by many [34]. Even worse, people may cherry-pick their generalization to create a false impression.

Example 1.1. *(March 4, 2020, Bernie Sanders) [4] "If you look at California, if you look at people of color in general, {African-Americans, Latinos, Asian-Americans,} we won that big time. Big time. Not even close." Democratic presidential candidate Sanders announced a big victory with people of color at a news conference. While this statement is technically correct, it suggests that he won among African-Americans, Latinos, Asian-Americans, etc., even if he hadn't explicitly named these sub-groups in the curly braces above. He includes African-Americans, which sub-group he lost to Joe Bidden by 20 points, in his claim about winning "people-of-color".*

Since African-Americans are a substantial sub-group among "people-of-color" in the U.S., we would expect a more balanced and informative statement to be that Sanders beat Biden among people of color in California, except African-Americans. On the other hand, suppose there was a small ethnic minority called "Purple-Americans", with only ten members in California. Even if Sanders had lost to Biden in this sub-group, we would care less if this exception was not mentioned in the generalization because of its minuscule size.

Errors in generalizing from the data occur not only as "spin" in the context of politically charged issues: they can even occur in "objective" arenas like science and medicine. For instance, doctors have over-diagnosed ADD and ADHD for years after making generalizations to age, sex, the maturity of the children [2].

Even if we base the generalized conclusion on aggregation over the entire data set, we could still fail to reach an appropriate representation of the situation, as is shown in the next example.

Example 1.2. *(IMDB) [23] According to IMDB, the television series* Sex and the City *is less popular than* The O.C *and* Gossip Girl, *which are all television series that came out in* 2000 − 2010, *with ratings of* 7.1, 7.5, *and* 7.4, *resp. This is surprising considering the popularity of* Sex and the City. *An explanation is that* Sex and the City *is particularly appealing to women. When considering the ratings provided solely by female raters, indeed,* Sex and the City *receives a rating of* 8, *compared with the* The O.C *and* Gossip Girl, *which get the ratings of* 7.5 *and* 7.7, *resp. However, male raters, give* Sex and the City *an average rating of 5.8. For most shows there may not be a significant difference between scores from men and women. When there is a large difference, as in this case, it may be essential to include that in the generalization produced. Without this, users are likely to have trouble interpreting the ratings appropriately.*

In the example above, the two sub-groups, male and female raters, had different opinions about a particular TV series. In light of these differences, a generalization over the sub-groups obscures

Proceedings of the VLDB Endowment, Vol. 15, No. 1 ISSN 2150-8097.
doi:10.14778/3485450.3485457

Table 1: Table of notations.

Symbol	Description
$f_Q(\mathcal{T})$	The statement derived by partition query Q
$Q^f(\mathcal{T})$	Groups that are relevant to statement $f_Q(\mathcal{T})$
$\mathcal{A}_{grp}, \mathcal{A}_{pred}$	Partition attributes for drill-down and slicing
r	Refinement expression given by $\mathcal{A}_r \subseteq \mathcal{A}_{pred}$
A	Attribute set $A \subseteq \mathcal{A}_{grp}$ as group-by predicate
Q^r_A	Refinement query with predicates r and A

the real underlying data and is therefore inappropriate. Another interesting application of generalization evaluation is the detection of the Simpson's paradox, which is a phenomenon that a trend of the whole population reverses within the sub-populations [9].

To determine whether a generalization is reasonable, one possibility is to consider how much support it has, evaluated as the proportion of pairwise individual comparisons supporting or opposing the statement. However, considering individual pairs may not give a meaningful explanation for the support, or lack thereof. A more natural way is to examine the refined sub-group comparisons and consider the size of the sub-groups as the weight to the support. Given generalizations derived from aggregate queries, we refine the queries using conjunctions of predicates to explore the responsible sub-groups as shreds of evidence for the support. In this way, an explanation for a low score would be a set of non-negligible sub-groups of the population opposing the given statement.

While a rich vein of research [17, 24–27, 36, 37] has been devoted to computational fact-checking aiming to evaluate the quality and correctness of statements based on structured data, most of them formulate the correctness of the statement based on the dataset query results. See related work (Section 6) for a brief summary. However, simply modeling the appropriateness of the generalization level via a single aggregate query or the perturbation of parameterized queries is not enough. As indicated in previous examples, misleading conclusions could still be derived if we only look at the aggregate results given by a single query. The evidence (opposing sub-groups) might depend on arbitrary combinations of predicates to validate such cherry-picked generalizations. We next outline our main contributions.

Statement scoring model (Section 2). We propose a model to quantify the quality of generalizations derived by aggregate queries. We refine the entities in the statements into sub-groups, through the notion of *refinement queries*. Then, using the refinement queries, we evaluate the data portion supporting/opposing the statement. Intuitively, the score of a statement reflects the degree to which the aggregation result represents the underlying data.

Efficient score evaluation (Section 3). The set of possible sub-groups (and the corresponding refinement queries) could be exponential with the number of attributes. Thus, a naive approach of enumerating all such queries to compute the statement score is inefficient. We propose an algorithm for efficient score computation that allows for the reuse of computational results, which is based on a hierarchy over refinement queries and a dedicated data structure.

Score explanation and statement refinement (Section 4). To provide the user a better understanding of the resulting score, we introduce the problem of providing *counterexamples*, i.e., disclosing significant parts of the data opposing the statement. We further discuss the problem of refining the statement to obtain alternatives that better represent the data. We present extensions to our score computation algorithm for identifying counterexamples and statement refinement utilizing an inverted index structure.

Table 2: Example data set.

Res ID	Gender	Age	Aged Over 35	Role	Salary
1	Male	25 - 34 years old	No	Developer	94K
2	Male	25 - 34 years old	No	DB Admin	11K
3	Female	25 - 34 years old	No	Developer	86K
4	Female	18 - 24 years old	No	Designer	19K
5	Male	35 - 44 years old	Yes	Developer	13K
6	Male	45 - 54 years old	Yes	DB Admin	18K
7	Female	35 - 44 years old	Yes	Developer	14K
8	Female	35 - 44 years old	Yes	Designer	40K

2 MODEL

We start by presenting our model for statements and the score using the notations of partition queries and query refinement. We demonstrate the ideas using the following running example. For simplicity, we consider a single table dataset. However, as we discuss in Section 7, our framework is also adaptable to more general scenarios. For convenience, we summarize the core symbols in Table 1.

Example 2.1. *(Ageism in Tech) Age discrimination towards people over 35 in the IT industry is a growing issue that has attracted much attention. Developers over 35 may feel over the hill when working in the tech industry [1]. We consider a projection of the Stack Overflow dataset, containing answers of users to the Stack Overflow developers survey, presented in Table 2. For simplicity, we consider only the Gender, Age (discretized), Role, and Salary attributes and construct an Aged Over 35 attribute to indicate the age group of each respondent. The aggregation over the example dataset suggests that respondents under 35 earn more on average compared with respondents over 35. However, this might not be an appropriate representation of the data because, as shown next, this statement might hold differently for people in different roles. We provide a detailed analysis in Section 5.1.*

2.1 Statements

We consider statements derived by aggregate value comparisons. We start by defining partition queries.

Definition 2.1 (Partition Query). *Let $\mathcal{T}$ be a table with a set of n attributes $\mathcal{T}_{\mathcal{A}} = \{A_1, \ldots, A_n\}$. A partition query Q is a GROUPBY aggregate query of the form*

```
SELECT attr, agg(target)
  FROM T
 WHERE cond
 GROUP BY attr
```

Where attr$\subseteq \mathcal{T}_{\mathcal{A}}$, target $\in \mathcal{T}_{\mathcal{A}}$, *and* cond *contains the WHERE clause conditions for $A_i \in \mathcal{T}_{\mathcal{A}}$.*

We denote by $Q.cond$ and $Q.attr$ the attribute sets in the cond expression and attr resp. , and $Q(\mathcal{T})$ to denote the query result.

Example 2.2. *Consider the dataset in Table 2. The following query is an example of a partition query Q.*

```
SELECT Aged_over_35, avg(Salary)
 FROM T
GROUP BY Aged_over_35
```

Here, $Q.cond = \emptyset$ (as no WHERE *clause), and $Q.attr$ = {*Aged_over_35*}.*

We assume an arbitrary order over the tuples in $Q(\mathcal{T})$ and use g_i to refer the i'th group in $Q(\mathcal{T})$, and $agg(g_i)$ to the aggregate result of g_i. Our model focuses on decomposable aggregate functions [14], which can be computed by the aggregate of subsets, examples include COUNT, MIN, MAX, SUM, AVERAGE, RANGE. Such queries are important in online analytical processing (OLAP) as they allow the computation to be performed on the pre-computed results instead of the base data. We consider COUNT aggregation for non-numeric target attributes or when the target attribute is not provided.

A statement over the database is based on a *comparison* of aggregate values of two or more selected groups obtained by a partition query Q. We aim to devise a unified mechanism for specifying various different types of statements.

Example 2.3. *Consider again the dataset $\mathcal{T}$ given in Table 2. Examples of statements with respect to $\mathcal{T}$ (and some partition queries) are: ($\mathrm{S_1}$) "The average salary of respondents aged under 35 is higher than that of respondents aged over 35" or ($\mathrm{S_2}$) "On average, developers earns over twice than of designers, and the salary of designers is higher by 50% that database administrators".*

We define a statement using a Boolean function as follows.

Definition 2.2 (Statement). *Let Q be a partition query, a statement $f_Q(\mathcal{T})$ is a Boolean function $f_Q : (\mathcal{T}) \to \{0, 1\}$ indicating the conditions that are satisfied by the aggregate value comparisons of some of the groups in $Q(\mathcal{T})$, obtained by the partition query Q. We say that a statement holds if the conditions are satisfied, giving $f_Q(\mathcal{T})$=1.*

We denote by $Q^f(\mathcal{T})$ the groups in $Q(\mathcal{T})$ which are relevant to the statement f.

Example 2.4. *Continuing with our running example, let g_1 denote the group associated with respondents aged under 35, and g_2 is the group associated with respondents aged over 35. The statement S_1 from Example 2.3, defining an order over the average salary of respondents in two age groups can be defined using the following function:*

$$f_Q(\mathcal{T}) = \begin{cases} 1, & \text{if } agg(g_1) - agg(g_2) \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Here, $Q^f(\mathcal{T})$ consists of two groups, aged under 35 and aged over 35, which are relevant for the function $f_Q(\mathcal{T})$.

2.2 Query Refinement

To explore the refined sub-group comparisons, we next define the notion of *query refinement.* We consider typical OLAP operations *drill-down* and *slice* [18] to get finer partitions of the groups. Drill-down steps down a concept hierarchy of a dimension, which in our setting can be performed by adding attributes to `Q.attr`. Slice picks a particular sub-set by choosing a single value for one of its dimensions, which in our setting be performed by adding conjunctions of attribute-value assignments to `cond`. Refining a partition query allows for the comparison of the aggregate results of sub-groups, which in turn can determine how well the underlying data is reflected by the partition query and reveal misleading conclusions. For instance, the statement $\mathbf{S_1}$ from Example 2.3 is supported by the query results given by Example 2.2, but does not reflect the fact that it is not the case for database administrators and designers. To this end, we define the sets of partition attributes $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$. We consider discretized/categorical attributes here as continuous attributes would give infinite sub-groups. The $\mathcal{A}_{grp}$ attributes are used to refine the `GROUP BY` clause given by `Q.attr`, and the $\mathcal{A}_{pred}$ attributes are used to refine the `WHERE` clause given by `Q.cond`.

Definition 2.3 (Query Refinement). *Given a partition query Q, and the attribute sets $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$, a refinement query Q^r_A is a query with: $Q.attr \subseteq Q^r_A.attr \subseteq \mathcal{A}_{grp} \cup Q.attr$, and $Q.cond \subseteq Q^r_A.cond$ $\subseteq \mathcal{A}_{pred} \cup Q.cond$.*

Example 2.5. *Let $\mathcal{A}_{pred}$= {`Gender, Role`} and $\mathcal{A}_{grp}$= {`Age`}, a possible refinement query Q^r_A of the query Q in Example 2.2 is:*

```
SELECT Aged_over_35, Age, avg(Salary)
  FROM T
 WHERE Gender = Female and Role = Developer
 GROUP BY Aged_over_35, Age
```

Here, Q^r_A.cond ={`Gender, Role`} and Q^r_A.attr ={`Aged_over_35, Age`}.

In what follows, when it is clear from the context, we refer to both partition queries and refinement queries simply as queries.

Partition attribute selection. We next describe the difference between $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$. The set $\mathcal{A}_{grp}$ contains attributes that are used to drill down the concept hierarchy given by $Q.attr$. For instance, in our running example, `Age` attribute is in $\mathcal{A}_{grp}$ to drill down the age concept hierarchy. It cannot be in $\mathcal{A}_{pred}$ because the age groups (under 35 and over 35) contain different age ranges, adding `Age` to the WHERE clause will get empty results. When comparing the sub-group aggregate values given by $\mathcal{A}_{grp}$, we allow cross-group comparison, i.e., comparing respondents under 35 aged $[18-24]$, $[25-34]$, to respondents over 35 aged $[35-44]$, $[45-54]$. Attributes that share (almost) the same data domain in all groups of $Q(\mathcal{T})$, are in $\mathcal{A}_{pred}$ for slicing. In the running example, all age groups contain both males and females. Thus `Gender` would be used in $\mathcal{A}_{pred}$ to refine the query. In this case, we compare only the aggregate values of females (and males) of each age group.

The attribute sets $\mathcal{A}_{pred}$, and $\mathcal{A}_{grp}$ can be defined by the end-user; however, we propose a default setting. Partition attributes that are in the same conceptual dimensions of $Q.attr$ are added to $\mathcal{A}_{grp}$. Attributes having unique values, e.g., `res ID` in Table 2, are ignored. The rest are added to $\mathcal{A}_{pred}$. We recommend discarding attributes in $\mathcal{A}_{pred}$ that have a strong correlation with $Q.attr$[1]. This is to minimize the skew refinements that give significantly different sizes of the sub-groups. For example, if there is a strong correlation between the `Role` and the `Aged_over_35` attributes, resulting in a case that there are few developers aged over 35 and the others are all aged under 35, such sub-group comparisons are sensitive to outliers and cannot provide strong evidence for statement evaluation.

2.3 Statement's Score

Given a statement $f_Q(\mathcal{T})$ and the attribute sets $\mathcal{A}_{pred}$ and $\mathcal{A}_{grp}$, our goal is to define the score of the statement, which measures how well it reflects the underlying data. To define the score of a statement, we iterate over all possible refinements of the partition query Q and consider the size of the sub-groups as their potential

[1]We can compute the Cramer's V correlation matrix to examine the association between the categorical variables.

influence on the score. To this end, we next define the *weight of a refinement query*, and the *support of a statement*.

The weight of a refinement query. Given the attribute set $\mathcal{A}_{pred}$ and a partition query Q, we define the notion of *refinement expression*, an expression containing attribute value assignments to be added to $Q.cond$.

Definition 2.4 (Refinement Expression). *Given the attribute set $\mathcal{A}_{pred}$, let $A_r \subseteq \mathcal{A}_{pred}$ be a subset of attributes from $\mathcal{A}_{pred}$. A refinement expression r is a conjunction of value assignments to attributes in A_r.*

We denote by $\mathcal{R}$ the set of all possible refinement expressions for the attributes in $\mathcal{A}_{pred}$.

Example 2.6. *Consider the partition query in Example 2.2, a possible subset of attributes from $\mathcal{A}_{pred}$ is A_r = `{Gender}`. A possible refinement expression is r = `{Gender = Male}`.*

Given a partition query Q and a refinement expression r, we denote by Q^r the refined query obtained by adding r to the `WHERE` clause. Let $|g_i|$ be the number of tuples in the i-th group of $Q^f(\mathcal{T})$. Given a refinement expression r, we use g_i^r to denote the lineage of g_i (with r being a filter), and by $|g_i^r|$ its number of tuples. Intuitively, the weight of a refinement is the proportion of the sub-groups that qualify.

Definition 2.5 (Refinement Query Weight). *Given a database $\mathcal{T}$, a statement $f_Q(\mathcal{T})$, and a refinement expression r, the weight of the refinement query Q^r w.r.t. the groups in $Q^f(\mathcal{T})$ is defined as:*

$$weight(Q^r) = \frac{\sum_{g_i \in Q^f(\mathcal{T})} |g_i^r|}{\sum_{g_i \in Q^f(\mathcal{T})} |g_i|}$$

If r is empty, we say that $weight(Q^r) = 1$.

Example 2.7. *Consider again the statement $f_Q(\mathcal{T})$ with the underlying partition query Q (presented in Example 2.2), and the refinement expression r= `{Gender = Male}`. The weight of Q^r is the fraction of male respondents. The weight of this refinement query is:*

$$weight(Q^r) = \frac{2 \text{ (male, Under 35)} + 2 \text{ (male, Over 35)}}{4 \text{ (Under 35)} + 4 \text{ (Over 35)}} = \frac{1}{2}$$

The support of a statement. Given a statement f_Q, and an attribute set $A \subseteq \mathcal{A}_{grp}$, we denote by Q_A the refinement query obtained by adding the attributes in A to $Q.attr$. By adding attributes into $Q.attr$, we partition each group $g_i \subseteq Q^f(\mathcal{T})$ into multiple subgroups $g_i^1, \ldots, g_i^s$. We then perform cross-group comparisons to compute the support. Intuitively, the support of a statement is the fraction of the sub-group comparisons that support the statement.

Let G_i^A denote the set of sub-groups obtained by partitioning the i-th group in the partition query output $Q^f(\mathcal{T})$ by adding A to $Q.attr$. To perform cross-group comparison for these sub-groups, the number of sub-group comparisons is $\prod_{i=1,\ldots,n} |G_i^A|$, where n is the number of groups in $Q^f(\mathcal{T})$.

Example 2.8. *Consider again our running example statement, where g_1 and g_2 refer to the groups of respondents under 35 and over 35 respectively. By adding the attribute set* `A = {Age}` *to $Q.attr$, g_1 is partitioned into $G_1^A = \{[18-24], [25-34]\}$ and $G_2^A = \{[35-44], [45-54]\}$. To compare all sub-groups in G_1^A to sub-groups in G_2^A, the number of sub-group comparisons is $2 \times 2 = 4$.*

Table 3: All possible refinement expressions (G: Gender, R: Role), the weight of their corresponding refinement queries, and the support for the two possible subsets of $\mathcal{A}_{grp}$, $A_1 = \emptyset$, A_2 = {Age}.

refinement expressions	$weight(Q^r)$	$supp(f_{Q^r_{A1}}(\mathcal{T}))$	$supp(f_{Q^r_{A2}}(\mathcal{T}))$
{}	1	1	0.75
{R = Developer}	0.5	1	1
{R = DB Admin}	0.25	0	0
{R = Designer}	0.25	0	0
{G = F}	0.5	1	0.75
{G = M}	0.5	1	0.5
{G = F, R = Developer}	0.25	1	1
{G = F, R = DB Admin}	0	0	0
{G = F, R = Designer}	0.25	0	0
{G = M, R = Developer}	0.25	1	1
{G = M, R = DB Admin}	0.25	0	0
{G = M, R = Designer}	0	0	0

Given a statement $f_Q(\mathcal{T})$, let $g_1^i \in G_1^A, \ldots, g_n^j \in G_n^A$ denote n sub-groups obtained by adding the attribute set A to $Q.attr$. We denote by $\mathcal{T}[g_1^i, \ldots, g_n^j]$ the database containing only the tuples belong to $\bigcup_{j=1}^{n} g_j^{i_j}$. The statement support is defined as follows.

Definition 2.6 (Statement support). *Given a statement $f_Q(\mathcal{T})$, and an attribute set $A \subseteq \mathcal{A}_{grp}$, the statement support w.r.t. Q_A is:*

$$supp(f_{Q_A}(\mathcal{T})) = \frac{\sum_{g_1^j \in G_1^A, \ldots, g_n^k \in G_n^A} f_{Q_A}(\mathcal{T}[g_1^j, \ldots, g_n^k])}{\prod_{i=1,\ldots,n} |G_i^A|}$$

where n is the number of groups in $Q^f(\mathcal{T})$.

Example 2.9. *As shown in Example 2.8, by adding the attribute set* `A = {Age}` *to $Q.attr$, we refine the groups g_1 and g_2 into two groups. The total number of sub-group combinations considered for the support computation is 4. The comparisons of different age ranges that satisfy the statement are* ⟨`[25-34]`, `[35-44]`⟩, ⟨`[25-34]`, `[45-54]`⟩ *and* ⟨`[18-24]`, `[45-54]`⟩. *Therefore, the support of the statement with respect to A is $\frac{3}{4} = 0.75$ (three out of four comparisons hold).*

The score of a statement. To define the score of a statement, we consider all refinement queries obtained by either modifying $Q.cond$ or $Q.attr$. Given a refinement expression r, and a set of attributes $A \subseteq \mathcal{A}_{grp}$, we use Q_A^r to denote the refinement query obtained by adding both r to $Q.cond$ and A to $Q.attr$. Finally, we define the score of a statement as follows.

Definition 2.7 (Statement Score). *Given a statement $f_Q(\mathcal{T})$, and the attribute sets $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$, the score of $f_Q(\mathcal{T})$ is:*

$$score(f_Q(\mathcal{T})) = \frac{\sum_{r \in \mathcal{R}} weight(Q^r) \cdot \left(\sum_{A \subseteq \mathcal{A}_{grp}} supp(f_{Q_A^r}(\mathcal{T})) \right)}{\sum_{r \in \mathcal{R}} \sum_{A \subseteq \mathcal{A}_{grp}} weight(Q_A^r)}$$

If $\mathcal{A}_{pred}=\emptyset$, then $\mathcal{R}=\emptyset$. In this case, we consider the entire population, thus $\sum_{r \in \mathcal{R}} weight(Q^r)=weight(Q)=1$. Then, the score of a given statement is defined only by its support value. If $\mathcal{A}_{grp}=\emptyset$, $\sum_{A \subseteq \mathcal{A}_{grp}} supp(f_{Q_A^r}(\mathcal{T}))=supp(f_{Q^r}(\mathcal{T}))=f_{Q^r}(\mathcal{T})$, which is either 0 or 1. We note that the score is normalized to the range of $[0, 1]$ by dividing by the sum of weights of all refinement queries.

Example 2.10. *Consider our running example statement, all possible refinement expressions r, subsets A of attributes from $\mathcal{A}_{grp}$, and their corresponding supports are depicted in Table 3. In the running example, the sum of weights of all refinement queries is 8. Following Definition 2.7, the normalized score is $score(f_Q(\mathcal{T})) = \frac{5.375}{8} = 0.672$.*

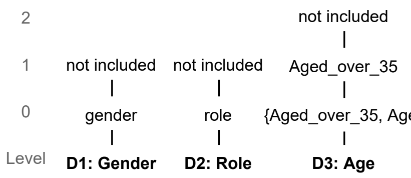


Figure 1: Dimension hierarchy.

Intuitively, the score reflects the population of sub-groups supporting the generalization. A higher score indicates a better reflection of the underlying data. In the running example, observe that although the generalized aggregation supports the statement, it still gets a relatively low score since a large fraction of sub-groups, such as designers and database administrators, opposing the statement.

3 SCORE COMPUTATION

We are now ready to define the statement validation problem.

Problem 1 (Statement validation). *Given a statement $f_Q(\mathcal{T})$, $\mathcal{A}_{pred}$ and $\mathcal{A}_{grp}$ compute $score(f_Q(\mathcal{T}))$.*

A naive algorithm for score computation would follow Definition 2.7 and operate as follows. Given a statement $f_Q(\mathcal{T})$, and the attribute sets $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$, iterate over all possible refinement expressions in $\mathcal{R}$. For each $r \in \mathcal{R}$ compute: (1) the weight of Q^r and (2) the support of $f_{Q^r_A}(\mathcal{T})$ for each $A \subseteq \mathcal{A}_{grp}$. Then using these results compute the score of $f_Q(\mathcal{T})$. Note that $|\mathcal{R}|$ is exponential in $|\mathcal{A}_{pred}|$ and the number of possible queries Q^r_A is exponential in $|\mathcal{A}_{grp}|$. To execute all queries for weight and support computation, the naive approach may lead to prohibitive execution time.

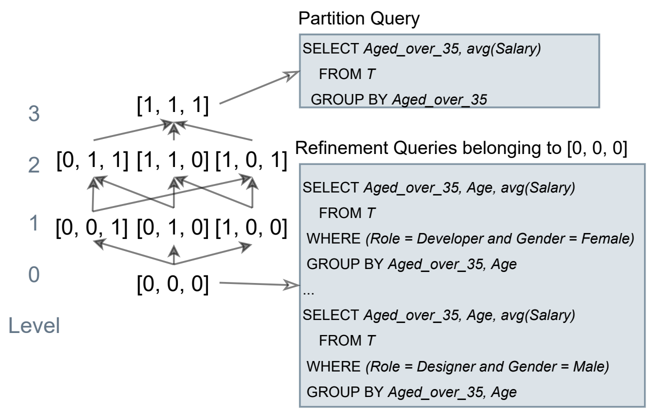
We next outline an improved algorithm that works well in practice, as we show in our experiments. The algorithm constructs a Hasse diagram based on the dimension hierarchy of the partition attributes, which represents a partial order over the refinement queries. Enumerating the refinement queries in a bottom-up fashion allows for the reuse of computational results (using a dedicated data structure). Thus we don't need to access the dataset to get aggregate results repeatedly, which reduces the running time significantly. We start by describing the refinement queries hierarchy.

3.1 Refinement Queries Hierarchy

We first define the notion of generalization level of refinement queries. In the OLAP multidimensional view of the data, attributes are grouped by their context into dimensions. In the i^{th} dimension, there is a hierarchy given by a set of attributes $\{A_{i_1}, \ldots, A_{i_m}\} \subseteq \mathcal{T}_\mathcal{A}$ to define hierarchical partitions of $\mathcal{T}$ into groups, where at the top level all the tuples are in a single group, and the lower levels reflect partitions at different granularities for the same aspect of the data.

Example 3.1. *The attributes set {*`Aged_over_35`*,* `Age`*} in Table 2 can be used to define a hierarchy over the age dimension. As is shown in Figure 1, the top level of the hierarchy groups together all the tuples in the data, the first level partition the tuples based on the* `Aged_over_35`*, and the bottom layer group together tuples with the same values in both, the* `Aged_over_35` *and* `Age` *attributes.*

Following common practice in OLAP, we assume a set of disjoint dimensions over partition attributes are provided by the user. When not provided, a default hierarchy of attributes singleton for each



```
SELECT Aged_over_35, avg(Salary)
  FROM T
 GROUP BY Aged_over_35
```

```
SELECT Aged_over_35, Age, avg(Salary)
  FROM T
 WHERE (Role = Developer and Gender = Female)
 GROUP BY Aged_over_35, Age
...
SELECT Aged_over_35, Age, avg(Salary)
  FROM T
 WHERE (Role = Designer and Gender = Male)
 GROUP BY Aged_over_35, Age
```

Figure 2: QRH for the running example ($\mathcal{P}_Q = [1, 1, 1]$).

attribute is used. The generalization level of a query is then defined by its dimension hierarchy levels.

Definition 3.1 (Generalization Level). *Given a query Q, let ℓ_i denotes the domain generalization level used for the dimension d_i in Q. The generalization level of Q is denoted by $\mathcal{P}_Q = [\ell_1, \ldots, \ell_k]$.*

Example 3.2. *In our running example, a possible dimension definition is* `D1:Gender`, `D2:Role`, *and* `D3:Age`. *Figure 1 shows the hierarchies for each dimension. The generalization level of the partition query Q (Example 2.2) is $\mathcal{P}_Q = [1, 1, 1]$, and the generalization level of the refinement query Q^r_A in Example 2.5 is $\mathcal{P}_{Q^r_A} = [0, 0, 0]$.*

Finally, we can define a partial order over the set of all queries.

Definition 3.2. *Given two queries Q and Q', we use $\mathcal{P}_Q \leq \mathcal{P}_{Q'}$ to denote that $\forall i, \mathcal{P}_Q[i] \leq \mathcal{P}_{Q'}[i]$. We say $\mathcal{P}_{Q'}$ dominates $\mathcal{P}_Q$ if $\mathcal{P}_Q \leq \mathcal{P}_{Q'}$.*

The definition of the generalization level provides a hierarchy over all queries that are dominated by the partition query Q. We construct a Hasse diagram to represent the partial order of the generalization levels. We refer to this diagram as the *Query Refinement Hierarchy* (QRH). Each node in QRH represents a generalization level that contains a set of queries. We say that the query $Q' \in v$ where v is a node in QRH if $\mathcal{P}_{Q'} = v$. We use $v_\perp$ to denote the bottom node in the diagram ($v_\perp = [0, 0, \ldots, 0]$). Figure 2 shows the QRH for the running example. As shown, each node corresponds to a set of refinement queries in the same level (with different predicate value assignments). All sub-groups given by queries in the same node are mutually exclusive, and their union is the entire population.

3.2 Algorithm

The key idea of our improved algorithm relies on the following observation. Given the count and the aggregate value of each group that belongs to some refinement query Q^r_A of Q, we can compute the weight and support of Q^r_A without executing the query. To this end, we define the Group Memorization (GM) table of a node $v \in$ QRH, that contains aggregate information for every refinement query $Q^r_A \in v$. Intuitively, our algorithm traverses the nodes of the QRH diagram in a bottom-up fashion, computes the GM table for each node, and uses it to calculate the weight and support of the refinement queries, *while scanning the data exactly once.*

Node	Refinement Expression	Grouping Attributes	Count	Agg value
[0,0,0]	{G = F, R = Developer}	{[25, 34], Under 35}	1	86K
	{G = F, R = Developer}	{[35, 44], Over 35}	1	14K
	{G = M, R = Developer}	{[25, 34], Under 35}	1	94K
	{G = M, R = Developer}	{[35, 44], Over 35}	1	13K
	{G = F, R = Designer}	{[18, 24], Under 35}	1	19K
	{G = F, R = Designer}	{[35, 44], Over 35}	1	40K
	{G = M, R = DB Admin}	{[25, 34], Under 35}	1	11K
	{G = M, R = DB Admin}	{[45, 54], Over 35}	1	18K
[1,0,0]	{R = Developer}	{[25, 34], Under 35}	2	90K
	{R = Developer}	{[35, 44], Over 35}	2	13.5K
	{R = Designer}	{[18, 24], Under 35}	1	19K
	{R = Designer}	{[35, 44], Over 35}	1	40K
	{R = DB Admin}	{[25, 34], Under 35}	1	11K
	{R = DB Admin}	{[45, 54], Over 35}	1	18K

Figure 3: Example GM tables of two nodes.

We next formally define the GM table of a node $v\in$ QRH and explain how it can be efficiently computed. We then present our optimized algorithms. For the simplicity of presentation, we assume a numeric target value and average as the aggregate function. Our algorithms can be adjusted to support other aggregate functions and non-numeric target values, as described in Section 2. We also assume the statements describe a total ordered comparison of the groups' aggregation values (partial ordered statements can be equivalently represented by several total ordered statements).

Definition 3.3 (The GM Table). *Let v be a node in QRH, and let GM_v denote the GM table of v. GM_v stores the values of $|g_i|$ and $agg(g_i)$, for every (non-empty) group $g_i \in Q^r_A(\mathcal{T})$, where Q^r_A is a refinement query that belongs to node v.*

Figure 3 depicts the GM table of the nodes $[0, 0, 0]$ and $[1, 0, 0]$ from the QRH diagram presented in Figure 2.

We can prove the following holds.

Proposition 3.1. *Let $v \in QRH$. If $v = v_\perp$, GM_v can be computed using a single data scan. Otherwise GM_v can be computed using $GM_{v'}$, where v' is a child node of v in QRH.*

Example 3.3. *Consider the GM tables given in Figure 3, and let $v = [1, 0, 0]$. A child node of v is $v_\perp$. To compute GM_v, we iterate over every group in $v_\perp$, and remove from each refinement expressionthe predicate of the Gender dimension. Records with the same refinement expressions and grouping attributes are merged, where the count is the sum of counts, and the aggregated value is the weighted average.*

Algorithm 1 computes the score of a statement. Given a statement $f_Q(\mathcal{T})$, it first set the *score* to be 0 (line 1). Then it traverses the refinement queries by their generalization level (lines 2–11), starting from the least general refinement queries (i.e., the queries in $v_\perp$). For each node v, the algorithm computes GM_v (lines 3–7). For the bottom node $v_\perp$, the GM table is computed using a procedure, which preform a single scan of the data (line 4). For a node $v \neq v_\perp$, GM_v is computed using a child node v', following Proposition 3.1 (lines 6–7). The node v' is obtained by setting ℓ_j to be $\ell_j - 1$, where ℓ_j is the first non-zero data dimension in v. Next, the algorithm iterates over all refinement queries that belong to the current node v. For each query, the algorithm computes its weight and the support using GM_v, and updates the score accordingly (lines 8–11).

Weight and support computation using GM tables. To compute the weight of a query $Q^r_A \in v$, we sum the counts of the groups that satisfy r, and divide by the sum of counts of all groups in v. A naive way to compute the support of Q^r_A, is to count the number of sub-group comparisons that satisfy the statement and divide it by the total number of sub-group comparisons.

```
Algorithm 1: Score Computation
    input  : A statement $f_Q(T)$, the attributes set $\mathcal{A}_{pred}$ and
             $\mathcal{A}_{grp}$, and the QRH diagram.
    output : The $score$ of $f_Q(T)$.
1   $score = 0$
    Let $L$ be an enumeration of QRH in a bottom-up fashion
2   foreach node $v$ in $L$ do
3       if $v = v_\perp$ then
4           $GM_v \leftarrow$ getButtomGM ($f_Q(T), \mathcal{A}_{pred}, \mathcal{A}_{grp}$)
5       else
6           $v' \leftarrow$ a child node of $v$
7           $GM_v \leftarrow$ getGMFromChild ($v', v$)
8       foreach partition query $Q^r_A \in v$ do
9           $weight \leftarrow$ getWeight ($GM_v, Q^r_A$)
10          $supp \leftarrow$ getSupp ($GM_v, Q^r_A$)
11          $score \leftarrow score + weight \cdot supp$
12  return $score$
```

```
Algorithm 2: Get Support
    input  : A node $v \in QRH$, it's $GM_v$ table, and a query
             $Q^r_A \in v$.
    output : The support value $supp(Q^r_A)$.
1   foreach $g_i \in Q^f(T)$ do
2       $agg[g_i] \leftarrow$ getAggValue ($Q^r_A$)
3       $agg[g_i].sort(ascending = True)$
4       $n_i \leftarrow length(agg[g_i])$
5   while $i_1 \leq n_1, i_2 \leq n_2, \cdots, i_k \leq n_k$ do
6       find $i_m$, s.t. $agg[g_m][i_m]$ is the current minimum value
7       $i_m \leftarrow i_m + 1$,
8       if $m = k$ then
9           $a_m \leftarrow i_m$
10      else
11          $a_m \leftarrow a_{m+1} + a_m$
12  return $\frac{a_1}{\prod_i n_i}$
```

Example 3.4. *Consider refinement query Q^r_A with $r=$* `{Gender = Female, Role = Developer}`*, $A=$* `{Aged_over_35, Age}`*, and $\mathcal{P}_{Q^r_A} = [0, 0, 0]$. The corresponding node of Q^r_A in QRH is $v = [0, 0, 0]$. The GM table of v is given in Figure 3. Following the above description, we get the weight of the refinement query $weight(Q^r_A) = \frac{2}{8}$. The number of sub-group comparisons is only one:* `{Age = [25, 34], Aged_over_35 = No}` *(with an aggregate value of 86K) and* `{Age = [35, 44], Aged_over_35 = Yes}` *(with an aggregate value of 14K), which supports the statement. Thus the support is $supp(Q^r_A) = \frac{1}{1} = 1$.*

Efficient support computation. The naive support computation (following Definition 2.6), requires the consideration of aggregate

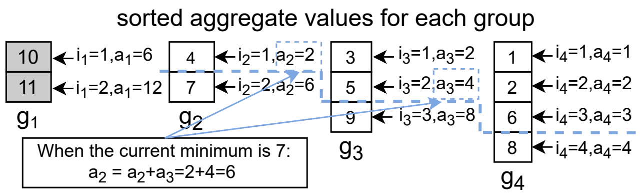


Figure 4: Efficient support computation.

value comparisons for all sub-groups in g_i for each $g_i \in Q^f(T)$. This makes the time complexity of `getSupp` in Algorithm 1 exponential in the number of groups in the statement $k=|Q^f(T)|$, i.e., $O(c^{k|A_{grp}|})$, where c is the highest attribute cardinality, and $c^{|A_{grp}|}$ is the greatest number of sub-groups in each group g_i. We present Algorithm 2 which is an optimized support computation algorithm reduces the complexity of `getSupp` to $O(kc^{|A_{grp}|}\log(c^{|A_{grp}|}))$. Intuitively, Algorithm 2 avoids unnecessary pair-wise comparisons of all aggregate values. It sorts the aggregate values for each considered sub-groups, iterates the sorted lists simultaneously, and examines only the "frontier" values.

W.l.o.g, we assume the total order in the statements is: $agg(g_1) > agg(g_2) > \cdots > agg(g_k)$. We compute a value a_j for each group g_j representing the number of value combinations for sub-groups from g_j to g_k that might satisfy the statement. The value a_1 would be the number of value combinations of all sub-groups that satisfy the statement. We next demonstrate our algorithm and explain the details using a visual example. Given a node v, its GM_v table, and a query $Q^r_A \in v$, the algorithm first computes and sorts the aggregate values for each group $g_i \in Q^f(\mathcal{T})$ (lines 1–4). The groups and their aggregate values are extracted from GM_v. Next, the algorithm iterates over the aggregate values (lines 5–11). It fetches the minimum value among all sorted aggregate value lists (line 6). For each newly fetched value, we first increase the index of this group i_m by 1 (line 7), and then we update the value a_m for this group (lines 8–11). If the minimum value is from group g_k, we set a_m to i_m, otherwise, a_m is updated by adding a_{m+1}. Finally, the algorithm returns the fraction of sub-group combinations that satisfy the statement.

In Figure 4, we show an example of computing the support for a statement containing four groups ($Q^f(T)$=4). The arrays are sorted aggregate values for each group. In each iteration, we show the i and a values. We first explain how a is updated. For example, when the current minimum is 7, the values we have enumerated are from 1 to 6. As shown, a_2 is updated by adding 2 to 4 which are the current values of a_2 and a_3. We next explain the meaning of the a values. a_1 is the number of value combinations that satisfy the statement. For example, when the current minimum is 10, a_1 is 6, representing the number of all combinations: (10,4,3,1), (10,4,3,2), (10,7,3,1), (10,7,3,2), (10,7,5,1),(10,7,5,2) that satisfy the statement.

Additional Optimizations. We conclude this section with two additional optimizations that could be applied on Algorithm 1. First, to compute the GM table of a node $v \neq v_\perp$, all we need is the GM table of one of its child nodes (Proposition 3.1). Thus, there is no need to keep in memory the GM tables of all nodes in QRH. It is enough to store the tables of nodes in only two levels at a time. Therefore, we track the current level i of the examined node. When moving on to a node in an upper level, we can delete all GM tables of nodes in level $i-1$. Second, we can use parallelism to speed up running times. According to our experiments, the "bottleneck" of the algorithm is when it iterates over all refinement queries of a given node v. We can therefore split all these queries into k batches, compute the score of each batch in parallel. Then, sum the k obtained score and move on to the next node. This is possible since the `getWeight` and `getSupp` procedures use only information from the GM table of v.

4 SCORE EXPLANATION AND STATEMENT CORRECTION

The score of a statement reflects the degree to which the result of the corresponding partition query represents the underlying data. Given a low score the user may wish to: (i) understand why the score is low (i.e., which parts of the data do not "agree with the statement") and (ii) refine the statement such that the new refined statement better represents the data but is "as close as possible" to the original query. In this section we formalize these two problems.

4.1 Counterargument Identification

Intuitively, sub-groups (of the groups considered by the statement) that do not align with the statement reduce the statement's score. Their sizes are used to quantify the effect on the score. Thus, an explanation of a low score may consist of these sub-groups. Counterarguments refer to the statements comparing these sub-groups but with a different conclusion to the original statement. For instance, a counterargument for (S_1) "The average salary of respondents aged under 35 is higher than that of respondents aged over 35" could be "The average salary of designers aged under 35 is lower than that of designers aged over 35". However, reporting all such statements can be overwhelming, as the number of opposing sub-groups may be exponential. In fact, it may obscure the "dominant" sub-groups. To this end, we focus on finding the set of counterarguments given by sub-groups that are not dominated by more general ones. We formally define the problem as follows.

Problem 2 (Counterargument Identification). *Given a statement $f_Q(\mathcal{T})$, find the set of statements $C = \{f_{Q'}(\mathcal{T}) \mid f_{Q'}(\mathcal{T}) = \neg f_Q(\mathcal{T}), Q' \preceq Q, and \nexists Q'' s.t. Q' \preceq Q'' \preceq Q and f_{Q''}(\mathcal{T}) = \neg f_Q(\mathcal{T})\}$.*

Recall that in the score computation, Algorithm 1 traverses over all refinement queries. Therefore, identifying the counterarguments can be done alongside the score computation using extra bookkeeping. One simple solution is to store a Boolean value for each refined statement, then traverse the QRH in a top-down breadth-first manner in search for counterarguments using the stored Boolean values. The search space may be pruned using the domination relationship of the refinement queries. Given the large number of refined statements to explore, the memory and time consumption for this naive solution could be prohibitive. We next provide a method to maintain the set of counterarguments C during score computation, hence avoiding this additional memory required and the QRH traversal.

Counterargument computation. To keep track of the set of counterarguments, we use the inverted index technique [20]. We augment Algorithm 1 with an additional inverted index V that stores the current set C. Figure 5 depicts the inverted index V for our running example. We store the counterarguments by their corresponding refinement queries. For each refinement query, we represent all predicates in form of value assignment A_i=a_i to represent

Predicates	Q_1: A_{pred}: Role = Designer g_1: {[18 − 24], Under 35} g_2: {[35 − 44], Over 35}	Q_2: A_{pred}: Role = DB Admin g_1: Under 35 g_2: Over 35	Q_3: A_{pred}: Role = Designer g_1: Under 35 g_2: Over 35
{Role = Developer}	0	0	0
{Role = Designer}	1	0	1
{Role = DB Admin}	0	1	0
{Aged_over_35 = No}	1	1	1
{Aged_over_35 = Yes}	1	1	1
{Age = [18 − 24]}	1	0	0
{Age = [35 − 44]}	1	0	0

Figure 5: Inverted index for counterargument computation.

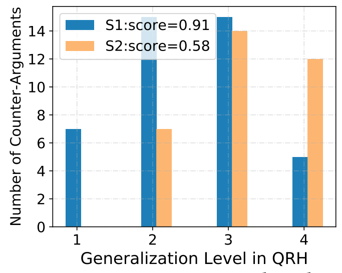


Figure 6: Counterargument distribution.

the refinement expression and the sub-group selection. The rows in V represent possible predicates for refinement queries, and the columns represent the current set of counterarguments C. The Boolean value $V[i][j]$ indicates whether the query Q_j contains the predicate component P_i, where P_i is a value assignment $A_k{=}v_b$.

During the bottom-up traversal of Algorithm 1, for each refinement query Q' and its corresponding sub-groups $Q'^f(\mathcal{T})$, s.t. $f_{Q'}(\mathcal{T}){=}\neg f_Q(\mathcal{T})$, C is updated. We first find the set of queries in C that are dominated by Q' via an efficient bitwise AND operation as follows. Let Q_j be the refinement query considered by the algorithm for which we find $f_{Q_j}(\mathcal{T}){=}\neg f_Q(\mathcal{T})$, and let $i_1 \ldots, i_k$ be the indices s.t. $V[p][j]{=}1$ for $p \in \{i_1 \ldots, i_k\}$. For example, in Figure 5, when adding Q_3, the corresponding indices are {2, 4, 5} (cells in dark gray) representing the predicates Q_3 contains. A query Q_m in C is dominated by Q_j if $V[p][m]{=}1$ for $p \in \{i_1 \ldots, i_k\}$. We use v_p to denote the vector of Boolean values in row p of the inverted index (before the insertion of Q_j). To efficiently compute the set of queries in C dominated by Q_j we apply a bitwise AND operation between the vectors v_p for $p \in \{i_1 \ldots, i_k\}$. The queries in C that are dominated by Q_j are the queries that are represented by the columns that correspond to the positions with the value 1 in the vector AND operation result. Continuing the example in Figure 5, to find the set of queries that are dominated by Q_3, we apply the AND operation between the three vectors $(1, 0)$, $(1, 1)$ and $(1, 1)$ that correspond to rows 2, 4 and 5 (and the columns of Q_1 and Q_2) (rows in gray). The result is $(1, 0)$ indicating that only Q_1 is dominated by Q_3. Lastly, columns that correspond to queries that are dominated by Q_j are removed from the inverted index and a new column (representing Q_j) is added.

Note that counterarguments can also be considered as statements, which could be cherry-picked as well. Given the resulting counterarguments, the user may further investigate them and compute their score using the score computation algorithm.

4.2 Statement Refinement

Given a statement $f_Q(\mathcal{T})$ and a threshold τ, if $score(f_Q(\mathcal{T})){\geq}\tau$, we call it a strong statement. Otherwise, we call it a weak statement.

Problem 3 (Statement Refinement). *Given a threshold τ and a weak statement $f_Q(\mathcal{T})$, find the set of statements $\mathcal{M}{=}\{f_{Q'}(\mathcal{T}) \mid f_{Q'}(\mathcal{T})$ is a strong statement, $Q'{\leq}Q$, and $\nexists Q''$ s.t. $Q'{\leq}Q''{\leq}Q$ and $f_{Q''}(\mathcal{T})$ is a strong statement$\}$.*

Given a weak statement $f_Q(\mathcal{T})$, finding alternative statements via statement refinement requires computing the scores for all queries Q' that are dominated by Q. A naive solution may apply a top-down breadth-first search over the QRH, utilizing Algorithm 1 to evaluate the score of $f_{Q'}(\mathcal{T})$ of each refinement query Q'. We next present a recursive score computation method using the space-time trade-off to avoid the repeated application of Algorithm 1.

The cumulative score computed during the refinement query enumeration in Algorithm 1 provides an easy way to compute the score for a given statement, but it does not reflect the score of the statement w.r.t. the refinement queries considered by the algorithm. We next present a recursive expression for score computation which can be used to compute the score of the statement w.r.t. the refinement queries alongside the execution of Alg. 1.

Proposition 4.1. *Let Q be a partition query, $v \in QRH$ such that $Q \in v$, $v' \in QRH$ a child node of v and $\overline{V'}$ the set of all descendent nodes of v in QRH excluding v' and its descendants.*

$$score(f_Q(\mathcal{T})) = \sum_{Q_i \in v'} \frac{weight(Q_i)}{weight(Q)} score(f_{Q_i}(\mathcal{T}))$$
$$+supp(f_Q(\mathcal{T})) + \sum_{v'' \in \overline{V'}} \sum_{Q_j \in v'', Q_j \leq Q} supp(f_{Q_j}(\mathcal{T}))$$

Example 4.1. *Consider the refinement query Q^r_A with r ={`Gender = Male`} and $A = \emptyset$, which belongs to the node $[0, 1, 1]$. To compute the score of the statement with respect to Q^r_A, we first fetch its children node $[0, 0, 1]$. The set of $\overline{V'}$ contains only one node $[0, 1, 0]$. In node $[0, 1, 0]$, we consider just the query Q'^r_A with r ={`Gender = Male`}, A = {`Age`} as it is the only query that satisfies $Q'^r_A \leq Q^r_A$. Therefore, the statement score is given by considering the scores of all queries in node $[0, 0, 1]$, and the support of Q'^r_A in node $[0, 1, 0]$.*

To find the set of alternative statements, we add two columns, `Score` and `Support`, to the GM tables, recording the score and support value of each refinement statement. This allows for alternative statements computation while executing Algorithm 1. In each step we use the recursive formula to compute the refinement statement scores. We use an inverted index structure, similar to Section 4.1, to maintain the current set of alternative statements.

5 EXPERIMENTAL STUDY

The first question to examine is whether our model creates "reasonable" scores for the appropriateness of generalization statements. We examine this question by in-depth look at examples, and let the reader judge for herself. We follow this with a quantitative experimental study to examine the efficiency of our algorithms in terms of running times and memory consumption in multiple practical scenarios. We evaluated our algorithms under varying number of partition attributes, data sizes, and number of groups in statements. We then evaluated the performance of the counterargument identification and statement refinement algorithms.

5.1 Proof of Concept

We next present two examples, demonstrating that our proposed framework provides a reasonable metric that intuitively reflects the sub-group fractions supporting/opposing real-life statements.

Ageism in Tech. [1] cited the StackOverflow yearly survey [3] that "among all participants, only 2% of developers are over 50, 4.6% are between ages of 40 to 50, and only 9.1% are between 35 and 39." The aggregate results strongly suggest an age barrier for developers over 35. However, the data show that is not all of the story. Here we look into two types of developers—front-end developers and engineering managers—to see how the conclusion holds based on the Stack Overflow (SO) dataset. We consider the following two statements S_1: *"The number of front-end developers under 35 is greater than front-end developers over 35"*, and S_2: *"The number of engineering managers under 35 is greater than engineering managers over 35"*.

We suggest considering partition attributes related to ethnicity, gender, education, and so on, which generate meaningful subgroups that users are interested in. Here we consider *{Age, RaceEthnicity, YearsCoding, FormalEducation}* as partition attributes. The four attributes belong to different concept dimensions; thus the query refinement hierarchy (QRH) contains five levels. Based on the definitions in Section 2, we consider the COUNT() aggregate function and $Q.att$= `Aged_over_35`, $Q.cond$= `Role`. The score for statement S_1 is 0.91 while the score of S_2 is only 0.58. To explain the underlying results for the difference between the relatively high and the relatively low score, we run the algorithms for counterargument identification in Section 4.1. As shown in Figure 6, for S_1, there are 42 counterexamples out of ~10K sub-groups explored, and most of these counterarguments are in lower levels of QRH. For example, S_1 might not hold for front-end developers from East Asia with a professional degree. On the other hand, for S_2, there are 33 counterarguments; most are in higher levels. For instance, S_2 does not hold for engineering managers who have been to graduate school (Master's and Ph.D.). A possible alternative for S_2 with a score threshold equals 0.85 is *"The number of engineering managers with less than 5 years coding experiences under 35 is greater than those over 35"*. The result indicates that the more significant the opposing sub-groups are, the less confident we are towards the statement.

Gender bias in academia. We next validate the utility of our framework in a real-life analysis of the gender gap in the promotion of faculty members. We use the dataset of the academic staff at a well-known university.[2] The dataset contains the information of over 1.2K academics hired by the university in the period 1990 – 2020. This information includes gender, rank (instructor, assistant professor, associate professor, or professor), research direction (art/life/science/law), employment type (full/part-time), and employment track. The data analysts are interested in investigating gender gaps in this hiring. We focus on the statement S : *the number of male researchers is greater than that of female researchers.* Here we use the count aggregation and consider the *{Rank, Direction, Employment type, Track}* as partition attributes. The score for the original statement is: 0.98 meaning male researchers generally dominate female researchers and the conclusion is unlikely to be cherry-picked. We next look into the total 6 counterexamples for score explanation, including (1) part-time instructors (weight =0.03, score =0.18), (2) part-time non-regular track researchers (weight=0.01, score =0.19), (3) non-regular track professors (weight =0.002, score =0), (4) part-time art researchers (weight =0.02, score =0), (5) non-regular track art researchers (weight =0.03, score =0.34), and (6) part-time life science researchers (weight =0.21, score =0.46). The above result shows that the counterarguments take up a very small fraction and have a low score under the conclusion of the original statement, which explains the relatively high score for statement S.

[2]Following the university request, the university's name is omitted and the data is private.

5.2 Experimental Setup

The experiments were implemented in Python 3.7 and executed on a Linux server with a 2.1GHz CPU, and 96GB memory.

Datasets and Statements. We used three datasets containing up to 2.4M records, which include multiple categorical attributes and can be associated with real-life statements.

Stack Overflow: Stack Overflow's (SO) annual Developer Survey is the largest survey of people who code around the world. The answers of the developers (i.e., the dataset) are publicly available [10]. It has more than 98K records containing information about the developers' age, place of residence, gender, ethnicity, income, and education. Besides the attributes city and country, each attribute in this dataset is related to a different data dimension. We enriched the dataset with additional attributes, such as age-group and continent (inferred from age and country, resp.), obtaining 11 dimensions with the maximal dimension hierarchy height of 3.

Police Killings: The Police Killings (PK) dataset is a publicly available dataset released by an online opinion poll website FiveThirtyEight [7], containing information regarding people killed by police and other law enforcement agencies in the United States throughout 2015 and 2016. We chose 11 attributes related to people demographics (e.g., age, gender, ethnicity) as well as details of the cases (e.g., cause of death and whether victims were armed or not).

US Census Data: The US Census dataset (USC) [13] is a discretized version of the 1990 US Census raw data, which contains information regarding the annual income and demographics of over 2.4M people in the US. From this dataset, we selected a subset of categorical attributes describing people demographics (such as the number of kids, marital status, and gender). We obtained 14 partition attributes with a maximal hierarchy height of 3.

The statements we derived range from Stack Overflow user reports [3] to news and media websites, including the Guardian [12], AlterNet[5], and the World Socialist Web Site[15]. They are used to generate insights about job opportunities, technology trends (Stack Overflow) or expose problems of contemporary society (Police Killings). For the US Census dataset, we generated synthesized statements uniformly at random. In all cases, attributes were assigned either to $\mathcal{A}_{pred}$ or $\mathcal{A}_{grp}$ as described in Section 2.2.

Baseline algorithms. To quantify the usefulness of our proposed algorithm for score computation, we assessed the effectiveness of each of the proposed optimizations and compared its results with the baseline solutions. We thus examined the following baselines:

Table 4: Default parameters.

	# of partition attributes	# of tuples	# of groups
Stack Overflow	5	98,855	2
Police killings	8	468	2
US Census Data	6	2,458,285	2

Naive. The naive algorithm, as described in Section 3.
Cube. The algorithm computes the aggregate values for all sub-groups using the CUBE operator [16] in PostgreSQL at the very beginning and runs the baseline algorithm using the cube.
Score Computation. Algorithm 1 with efficient support computation (Algorithm 2) without parallelism and memory optimization.
Optimized. The algorithm that includes all optimizations.

Parameter Setting. We examined the effect of the following parameters: (1) the number of attributes in $\mathcal{A}_{grp}$ and $\mathcal{A}_{pred}$; (2) the number of tuples in the datasets; (3) the number of groups compared in the statements. In each experiment, we varied the value of one parameter, while setting the others to the default value. The default values are depicted in Table 4. Some values were chosen to be the largest possible values in which all baseline algorithms can handle. The time cutoff used in all experiments was 90 minutes. The number of threads for parallelism was set to 20 or 4.

5.3 Score Computation

We found that for each dataset, the results among different statements demonstrated similar trends. Thus, we reported the average results obtained for three statements in each dataset.

Partition Attributes. We incrementally added partition attributes to $\mathcal{A}_{grp} \cup \mathcal{A}_{pred}$, comparing the execution time and memory overhead of different algorithms. For execution time, the results are depicted in Figures 7a–7c. The x-axis is the number of partition attributes, and the y-axis is the runtime.We observed a near-exponential growth in the running times as increasing the number of partition attributes. This is because the time complexity of all algorithms is exponential in $|\mathcal{A}_{grp}|$ and $|\mathcal{A}_{pred}|$. However, our model can still work for wide datasets because attributes not in $\mathcal{A}_{grp} \cup \mathcal{A}_{pred}$ would not affect the execution time as the sizes of the GM tables are only related to the number of partition attributes. The result indicates that there is a limitation of the number of partition attributes. This limitation exists in all OLAP cube computations; data systems like PostgreSQL usually limit the data cube to 12 attributes [28]. We note that this limitation has little effect on the availability of our model as sub-groups defined by more than 12 predicate combinations are typically very small (take up a very small percentage of the data population) and have a low impact on the result.

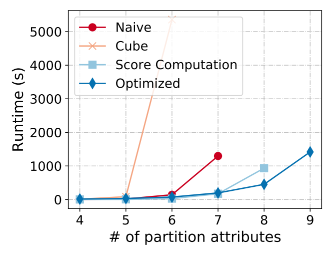
We use the Telco Customer Churn dataset [11], a fictional dataset with multiple low-cardinality attributes, to demonstrate the scalability of the algorithm runtime w.r.t. the number of partition attributes.[3] As shown in Figure 10, the *Score Computation* and *Optimized* can support up to 12 partition attributes and the execution time is exponential to the number of partition attributes. Nonetheless, in Figures 7a–7c, while the baseline algorithms can only handle a small number of attributes in reasonable running times, the *Score Computation* and *Optimized* algorithms show a better time efficiency and are more scalable as the number of attributes increases. With a small number of partition attributes, which leads to few refinement queries to explore (i.e., in very simple cases), the *Score Computation* algorithm performs better than the *Optimized* algorithm. This stems from the fact that in such cases, dividing the queries and unifying the results into a single score is wasteful. When the dataset is small (e.g., PK dataset), the execution time for the *Naive* algorithm is close to the *Score Computation* algorithm. This is because the time overhead of executing the queries on very small datasets is low; thus, the effect of the GM table is negligible.

The memory consumption for all algorithms is exponential to the number of partition attributes. In Figure 7d, we report the memory consumption under the maximal number of partition attributes for each dataset. The *Optimized* algorithm reduces memory consumption of the *Score Computation* algorithm by more than half. The GM tables also have smaller memory consumption compared with the *Cube* algorithm for most of the time as it resembles an incremental way to compute the cuboids of the cube. The GM tables only contain the predicate information in the corresponding refinement queries. There could be exceptions in computing some aggregate functions like AVERAGE, which needs auxiliary information, i.e. the sum and the count for each sub-group to compute the average. The results demonstrate that our algorithms can handle millions of refinement queries with low memory requirements.

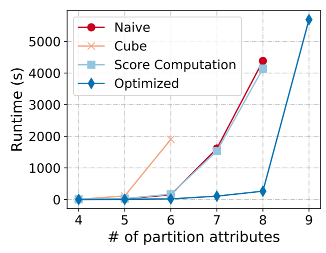
Dataset size. We examined the effect of the data size on the performance. To this end, we selected increasing portions of the datasets. For the small datasets (SO and PK), we further increased their size by generating additional tuples uniformly at random. The results are depicted in Figure 8, showing the effect on running times and memory consumption. Observe that the number of tuples has almost no effect on the *Score Computation* and the *Optimized* algorithms, while the running time of the *Naive algorithm* is linear to the number of tuples. This is due to the usage of the GM table. While the *Naive* approach queries the database for each weight and support computation, our algorithms query the database only once. This highlights the usefulness of the GM table when scaling up the data size, as the number of possible groups in the GM table is bounded by the number of attribute-value combinations (and not affected by the data size). The execution time of the *Cube* algorithm is also not affected by the data size, as it precomputes the aggregate data and uses it for score computation. There is an exception on the (small) PK dataset because the increase in the data size causes a drastic increase in the data cube size. However, the execution time for *Cube* algorithm would be prohibitive when the number of partition attributes is large. In this case, the size of the data cube swells, and it would take a longer time to fetch the aggregate results. For example, the cube contains 76,457 records in the PK dataset, while the dataset only has 467 rows. We next consider Figure 8d which shows the memory consumption under the maximal number of tuples. Observe that the memory consumption remains small because the sizes of GM tables and data cubes are only affected by the number and the cardinalities of partition attributes.

Number of groups. Last, we varied the number of groups to be compared and report the effect on running times and memory usage. This experiment set aims to evaluate the contribution of our

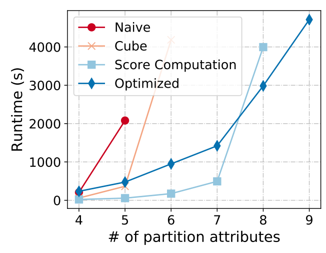
[3]Due to space constraints, the results of other experiments on the Telco Customer Churn dataset are omitted as they showed similar trends to the rest of the datasets.



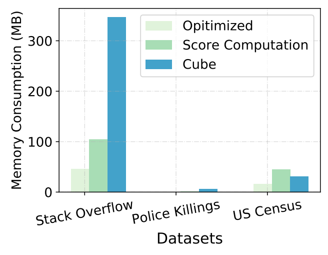
(a) Stack Overflow



(b) Police Killings

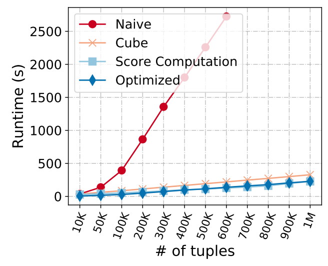


(c) US Census

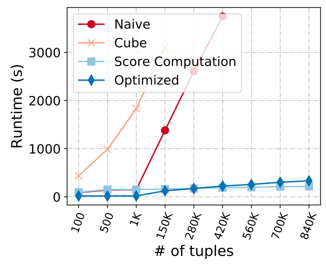


(d) Maximal memory consumption

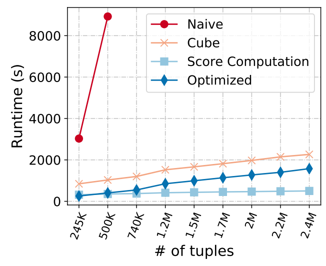
Figure 7: Runtime and memory consumption, varying the number of partition attributes.



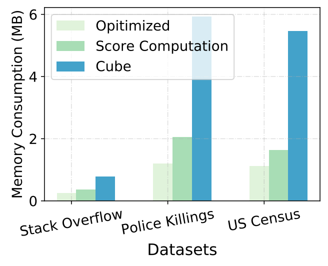
(a) Stack Overflow



(b) Police Killings

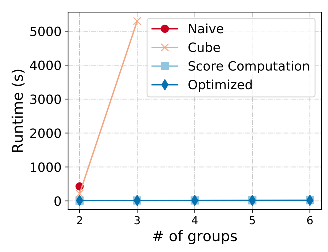


(c) US Census

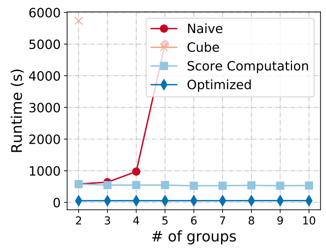


(d) Maximal memory consumption

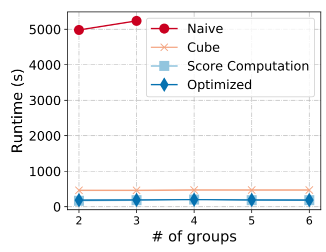
Figure 8: Runtime and memory consumption, varying data size.



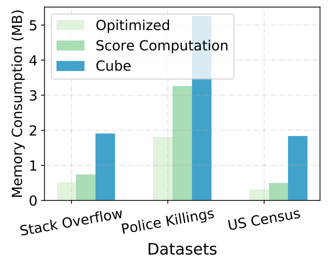
(a) Stack Overflow



(b) Police Killings



(c) US Census



(d) Maximal memory consumption

Figure 9: Runtime and memory consumption, varying the number of groups in statements.

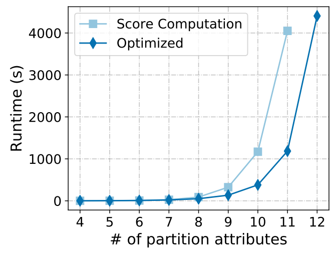


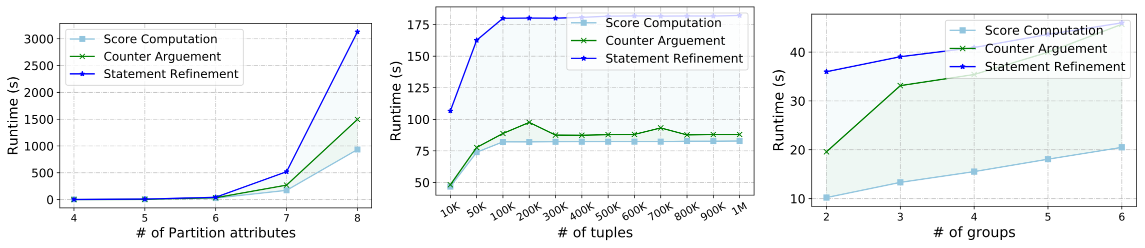
Figure 10: Algorithm runtime, varying the number of partition attributes (Telco Customer Churn dataset).

optimized support score computation (Algorithm 2). The results are depicted in Figure 9. Observe that the number of groups has very little impact on the execution times of the *Score Computation* and *Optimized* algorithms. Although the time complexity indicates that it is linear to the number of groups (Figure 11c has a clear view)), the trend is not obvious in most of the cases as the time to fetch the aggregate values dominates the time to sort and enumerate the values. In contrast, the running times of algorithms that use a naive approach for support computation (i.e., *Naive* and *Cube*) are exponential in the number of groups. The trend of *Cube* algorithm in the USC dataset is not obvious since the number of sub-groups in each group is small in this case. This is in accord with our complexity analysis. In Algorithm 2, we reduce the exponential time complexity of pairwise comparison to linear. In all cases, our optimized algorithm uses approximately half of the memory space compared with the algorithm that uses no memory pruning (Figure 9d).

5.4 Counterargument & Statement Refinement

Next, we study the performance of counterargument identification and statement correction algorithms. As all datasets indicate similar trends, we reported only the results of the SO dataset here. To identify counterarguments and refine the statements, we maintain an extra data structure while executing the algorithm for score computation. The maintenance of the inverted index structure cannot be done in parallel, therefore, in this section, we discuss the memory and time overhead of computing the counterarguments and refined statements compared with the *Score Computation* algorithm.

Figure 11 shows the execution time of the *counterargument identification* and the *statement refinement* algorithm under varying numbers of partition attributes, data size, and the number of groups in statements. Compared with the *Score Computation* algorithm, these two algorithms demonstrate the same tendency in the scaling experiments. Specifically, as the number of partition attributes



(a) Varying # of partition attributes **(b)** Varying data size **(c)** Varying # of groups

Figure 11: Runtime of counterargument identification and statement refinement (Stack Overflow).

increases, the execution time for all algorithms increases exponentially (Figure 11a). The size of the data set does not affect the execution time of the algorithms when it reaches a certain value (Figure 11b). This is due to our optimization using GM tables, which records the aggregate information for each sub-group to avoid accessing the data sets. The execution time is linear in the number of groups in the statements (Figure 11c). In all experiments, we showed that the time overhead for *counterargument identification* and *statement refinement* is reasonable. We also recorded the memory consumption for the inverted index structures. As they are all less than 3MB, we omitted the graphs from the presentation.

6 RELATED WORK

Computational fact-checking is an emerging research field [17, 24–27, 36–38]. Traditional fact-checking methods relying on the domain knowledge of human experts and crowdsourcing [6, 8, 21, 22] are not scalable and unconvincing without the presence of supporting datasets. A large body of work focuses on automating the fact-checking process with structured data. [36, 37] explore data perturbation and provide an efficient framework to model claims as parameterized queries for robustness and accuracy evaluation. [24–27] aim at checking aggregate data summaries from relational databases with a natural language interface. Our work not only focuses on maliciously false claims but also on the cherry-picking scenario, where the aggregate results could be true but misleading. [17] focuses on evaluating cherry-picked trendlines, where "unreasonable" trends could be derived from falsely chosen endpoints. To the best of our knowledge, no prior work has focused on the cherry-picked generalizations [33], where inductive conclusions are made based on insufficient evidence or cherry-picked aggregation levels.

The tests of statistical significance are common methods to evaluate the hypotheses about statistics of given groups. We distinguish our framework from these tests in the following aspects. First of all, our score and the p-value in the statistical significance tests have different meanings. The p-value is not a measure of the appropriateness of generalization levels. Tests of statistical significance assume the data from each group follows the same distribution and use the p-value to indicate if the observed data is statistically significant. Our score, in comparison, evaluates whether this assumption, which generalizes the data from various sub-groups, is reasonable. In addition, the tests of statistical significance focus on hypotheses of sample statistics, e.g., mean and standard deviation. It does not support many aggregate values, e.g. the COUNT aggregation. Our statement model supports more flexible comparisons and allows the users to specify conditions for comparison results.

Our work is closely related to multidimensional data aggregation. The multidimensional view of data is one of the most popular conceptual models of data warehousing. The *drill-down* and *rollup* operators tend to increase/decrease the levels of aggregation along dimension hierarchies [18], and the RELEX [30] operator inspects several kinds of generalization to get interpretable data summary. [29] focuses on the interactive exploration of data cubes and helps users to find the most informative and interesting parts of the data to explore. However, none of these works consider whether a generalization is a fair representation of the data. Our work proposes efficient algorithms to evaluate statements derived from aggregate value comparisons. Our computation of the GM tables also shares some similar ideas with the CUBE [16] operator to compute group-bys on all possible combinations of selected attributes. Experimental results indicate that the GM tables achieve better efficiency compared with using data cubes. The idea of enumerating the predicate combinations for aggregate values is also similar to Scorpion [35] which uses aggregate queries to explain away problematic data points, and SEEDB [32] explores aggregate data visualization.

7 CONCLUSION

In this paper, we provide a metric to determine if a statement given by generalized aggregation is a reasonable representation of the underlying data. We consider generalization evaluation in the context of fact-checking while recognizing a broader application of evaluating OLAP aggregations, which are essential components in many decision-support applications. For example, in exploratory data analysis [31], and data visualization [19]. We define the notion of *query refinement* to compute a cherry-picked score based on the sub-group proportion supporting/opposing the statement. We propose efficient algorithms for score computation, allowing the reuse of computational results and reduce time complexity; we also develop algorithms for identifying significant counterexamples and alternative statements to provide explanations of the score. Our framework and algorithms are not limited to single-table relational databases. Since the computation only relies on the GM tables, as long as the aggregate values are accessible, our framework and algorithms would be effective. Extensive experiments on real-world statements and datasets validate the effectiveness and efficiency of our proposed methods.

ACKNOWLEDGMENTS

This research was supported in part by NSF under grants 1741022 and 1934565 and was partially funded by the Israel Science Foundation, the Binational US-Israel Science Foundation, and the Tel Aviv University Center for AI and Data Science.

REFERENCES

[1] 2019. What happens to developers once they reach 35? https://blog.pitchme.co/2019/10/29/what-happens-to-developers-once-they-reach-35/.

[2] 2019. What to know about ADHD misdiagnosis. https://www.medicalnewstoday.com/articles/325595#age-related-factors.

[3] 2020. 2020 Stackoverflow Developer Survey. https://insights.stackoverflow.com/survey/2020.

[4] 2020. Fact Check: Did Bernie Sanders Win 'People Of Color' In California, And Was It 'Not Even Close'? https://www.politifact.com/factchecks/2020/mar/09/bernie-sanders/fact-check-did-bernie-sanders-win-people-color-cal/.

[5] 2021. AlterNet. https://www.alternet.org/2018/12/about-alternet/.

[6] 2021. FactCheck.org. https://www.factcheck.org/.

[7] 2021. Police Killings Dataset. https://github.com/fivethirtyeight/data/tree/master/police-killings.

[8] 2021. PolitiFact. https://www.politifact.com/.

[9] 2021. Simpson's paradox. https://en.wikipedia.org/wiki/Simpson%27s_paradox.

[10] 2021. Stack Overflow developer survey. https://insights.stackoverflow.com/survey.

[11] 2021. Telco Customer Churn Dataset. https://www.kaggle.com/blastchar/telco-customer-churn.

[12] 2021. TheGuardian.com. https://www.theguardian.com/us-news/series/counted-us-police-killings.

[13] 2021. US Census Data (1990) Data Set. https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990).

[14] 2021. WIKIPEDIA: Aggregate function. https://en.wikipedia.org/wiki/Aggregate_function#Decomposable_aggregate_functions.

[15] 2021. World Socialist Web Site. https://www.wsws.org/en/special/pages/icfi/wsws.html.

[16] Sameet Agarwal, Rakesh Agrawal, Prasad M Deshpande, Ashish Gupta, Jeffrey F Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. 1996. On the computation of multidimensional aggregates. In *VLDB*, Vol. 96. VLDB Endowment, 506–521.

[17] Abolfazl Asudeh, Hosagrahar Visvesvaraya Jagadish, You Wu, and Cong Yu. 2020. On detecting cherry-picked trendlines. *PVLDB* 13, 6 (2020), 939–952.

[18] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *SIGMOD* 26, 1 (1997), 65–74.

[19] Chun-houh Chen, Wolfgang Karl Härdle, and Antony Unwin. 2007. *Handbook of data visualization*. Springer Science & Business Media.

[20] Doug Cutting and Jan Pedersen. 1989. Optimization for dynamic inverted index maintenance. In *SIGIR*. ACM, 405–411.

[21] Naeemul Hassan, Fatma Arslan, Chengkai Li, and Mark Tremayne. 2017. Toward automated fact-checking: Detecting check-worthy factual claims by claimbuster. In *SIGKDD*. ACM, 1803–1812.

[22] Naeemul Hassan, Chengkai Li, and Mark Tremayne. 2015. Detecting check-worthy factual claims in presidential debates. In *CIKM*. ACM, 1835–1838.

[23] Walt Hickey. 2016. Men Are Sabotaging The Online Reviews Of TV Shows Aimed At Women. https://fivethirtyeight.com/features/men-are-sabotaging-the-online-reviews-of-tv-shows-aimed-at-women/.

[24] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Aggchecker: A fact-checking system for text summaries of relational data sets. *PVLDB* 12, 12 (2019), 1938–1941.

[25] Saehan Jo, Immanuel Trummer, Weicheng Yu, Xuezhi Wang, Cong Yu, Daniel Liu, and Niyati Mehta. 2019. Verifying text summaries of relational data sets. In *SIGMOD*. ACM, 299–316.

[26] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: A Mixed-Initiative Approach to Large-Scale, Data-Driven Claim Verification. *PVLDB* 13, 12 (2020), 2508–2521.

[27] Georgios Karagiannis, Mohammed Saeed, Paolo Papotti, and Immanuel Trummer. 2020. Scrutinizer: fact checking statistical claims. *PVLDB* 13, 12 (2020), 2965–2968.

[28] Babak Salimi, Johannes Gehrke, and Dan Suciu. 2018. Bias in olap queries: Detection, explanation, and removal. In *SIGMOD*. ACM, 1021–1035.

[29] Sunita Sarawagi. 2001. User-cognizant multidimensional analysis. *The VLDB Journal* 10, 2 (2001), 224–239.

[30] Gayatri Sathe and Sunita Sarawagi. 2001. Intelligent rollups in multidimensional OLAP data. In *VLDB*, Vol. 1. 531–540.

[31] John W Tukey et al. 1977. *Exploratory data analysis*. Vol. 2. Reading, Mass.

[32] Manasi Vartak, Sajjadur Rahman, Samuel Madden, Aditya Parameswaran, and Neoklis Polyzotis. 2015. SEEDB: Efficient Data-Driven Visualization Recommendations to Support Visual Analytics. *PVLDB* 8, 13 (2015), 2182–2193.

[33] Douglas Walton. 1999. Rethinking the fallacy of hasty generalization. *Argumentation* 13, 2 (1999), 161–182.

[34] John Woods. 2004. Hasty generalization. In *The Death of Argument*. Springer, 311–334.

[35] Eugene Wu and Samuel Madden. 2013. Scorpion: Explaining Away Outliers in Aggregate Queries. *PVLDB* 6, 8 (2013), 553–564.

[36] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2014. Toward computational fact-checking. *PVLDB* 7, 7 (2014), 589–600.

[37] You Wu, Pankaj K Agarwal, Chengkai Li, Jun Yang, and Cong Yu. 2017. Computational fact checking through query perturbations. *TODS* 42, 1 (2017), 1–41.

[38] Xinyi Zhou and Reza Zafarani. 2018. Fake news: A survey of research, detection methods, and opportunities. *arXiv preprint arXiv:1812.00315* 2 (2018).