Explaining Missing Answers to Top-k SQL Queries

Wenjian Xu, Zhian He, Eric Lo, and Chi-Yin Chow, Member, IEEE

Abstract—Due to the fact that existing database systems are increasingly more difficult to use, improving the quality and the usability of database systems has gained tremendous momentum over the last few years. In particular, the feature of explaining why some expected tuples are missing in the result of a query has received more attention. In this paper, we study the problem of explaining missing answers to top-k queries in the context of SQL (i.e., with selection, projection, join, and aggregation). To approach this problem, we use the query-refinement method. That is, given as inputs the original top-k SQL query and a set of missing tuples, our algorithms return to the user a refined query that includes both the missing tuples and the original query results. Case studies and experimental results show that our algorithms are able to return high quality explanations efficiently.

Index Terms—Missing answers, Top-K, SQL, usability

1 INTRODUCTION

AFTER decades of effort working on database performance, recently the database research community has paid more attention to the issue of database usability, i.e., how to make database systems and database applications more user friendly? Among all the studies that focus on improving database usability (e.g., keyword search [1], form-based search [2], query recommendation [3] and query auto-completion [4]), the feature of explaining why some expected tuples are missing in the result of a query, or the so-called “why-not?” feature [5], is gaining momentum.

A why-not question is being posed when a user wants to know why her expected tuples do not show up in the query result. Currently, end users cannot directly sift through the dataset to determine “why-not?” because the query interface (e.g., web forms) restricts the types of query that they can express. When end users query the data through a database application and ask “why-not?” but do not find any means to get an explanation through the query interface, that would easily cause them to throw up their hands and walk away from the tool forever—the worst result that nobody, especially the database application developers who have spent months to build the database applications, want to see. Unfortunately, supporting the feature of explaining missing answers requires deep knowledge of various database query evaluation algorithms, which is beyond the capabilities of most database application developers. In view of this, recently, the database community has started to research techniques to answer why-not questions on various query types. Among them, a few works have focused on answering why-not questions on Select-Project-Join-Aggregate (SPJA) SQL queries (e.g., [5], [6], [7], [8]) and preference queries (e.g., top-k queries [9], reverse skyline queries [10]). So far, these proposals only work independently. For example, when answering why-not questions on top-k queries, the proposal in [9] assumes there are no SP constructs (e.g., selection, projection, join, and aggregation).

In this paper, we study the problem of answering why-not top-k questions in the context of SQL. Generally, a top-k query in SQL appears as:

```
SELECT A1, ..., An, agg(·)
FROM T1, ..., Tk
WHERE P1 AND ... AND Pn
GROUP BY A1, ..., Am
ORDER BY f(·)
LIMIT k
```

where each $P_i$ is either a selection predicate “$A_i \, op \, v$” or a join predicate “$A_i \, op \, A_k$”, where $A_i$ is an attribute, $v$ is a constant, and $op$ is a comparison operator. Besides, $agg(·)$ is an aggregation function and $f(·)$ is a ranking function with weighting vector $\vec{w}$. The weighting vector represents the user preference when making multi-criteria decisions.

To address the problem of answering why-not questions on top-k SQL queries, we employ the query refinement approach [8], [9], [10]. Specifically, given as inputs the original top-k SQL query and a set of missing tuples, this approach requires to return to the user a refined query whose result includes the missing tuples as well as the original query results. In this paper, we show that finding the best refined query is actually computationally expensive. Afterwards, we present efficient algorithms that can obtain the best approximate explanations (i.e., the refined query) in reasonable time. We present case studies to demonstrate our solutions. We also present experimental results to show that our solutions return high quality explanations efficiently. This paper is an extension of [9], [11], which...
discussed answering why-not questions on top-k queries in the absence of other SQL constructs such as selection, projection, join, and aggregation.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents (i) the problem formulation, (ii) the problem analysis, and (iii) the algorithms of answering why-not questions on top-k SQL queries. Section 4 extends the discussion to top-k SQL queries with GROUP BY and aggregation. Section 5 demonstrates both the case studies and the experimental results. Section 6 concludes the paper.

2 RELATED WORK

Explaining a null answer for a database query was set out by [12], [13] but the concept of why-not was first formally discussed in [5]. That work answers a user’s why-not question on Select-Project-Join (SPJ) queries by telling her which query operator(s) eliminated her desired tuples. After that, this line of work has gradually expanded. In [6] and [7], the missing answers of SPJ [6] and SPJA [7] queries are explained by a data-refinement approach, i.e., it tells the user how the data should be modified (e.g., adding a tuple) if she wants the missing answer back to the result. In [8], a query-refinement approach is adopted. The answer to a why-not question is to tell the user how to revise her original SPJA queries so that the missing answers can return to the result. They define that a good refined query should be (a) similar — have few “edits” comparing with the original query (e.g., modifying the constant value in a selection predicate is a type of edit; adding/removing a join predicate is another type of edit) and (b) precise — have few extra tuples in the result, except the original result plus the missing tuples. In this paper, we adopt the query-refinement approach as our explanation model and also apply the above similarity and precision metrics.

This paper is an extension of our early work [9], [11], which discussed the answering of why-not questions on top-k queries in the absence of other SQL constructs such as selection, projection, join, and aggregation. Considering these additional SQL constructs could complicate the solution space. As an example, consider table $U$ in Fig. 1a and the following top-3 SQL query:

$$Q_0: \begin{align*} &\text{SELECT U.ID} \\ &\text{FROM U} \\ &\text{WHERE U.A \geq 205} \\ &\text{ORDER BY 0.5 * U.A + 0.5 * U.B} \\ &\text{LIMIT 3} \end{align*}$$

Fig. 1b shows the ranking scores of all tuples in $U$ and the top-3 result is [P3, P1, P2]. Assuming that we are interested in asking why P5 is not in the top-3, we see that using SPJA query modification techniques in [8] to modify only the SPJ constructs (e.g., modifying WHERE clause to be $U.A \geq 140$) cannot include P5 in the top-3 result (because P5 indeed ranks 4th under the current weighting $\bar{w} = [0.5, 0.5]$). Using our preliminary top-k query modification technique [9] to modify only the top-k constructs (e.g., modifying $k$ to be four) cannot work either because P5 is filtered by the WHERE clause. This motivates us to develop holistic solutions that consider the modification of both SPJA constructs and top-k constructs in order to answer why-not questions on top-k SQL queries. For the example above, the following refined query $Q'$ is one candidate answer:

$$Q': \begin{align*} &\text{SELECT U.ID} \\ &\text{FROM U} \\ &\text{WHERE U.A \geq 140} \\ &\text{ORDER BY 0.5 * U.A + 0.5 * U.B} \\ &\text{LIMIT 4} \end{align*}$$

$Q'$ is precise because it includes no extra tuple and is similar to $Q_0$, because only essential edits were carried out: (1) modifying from $U.A \geq 205$ to $U.A \geq 140$, and (2) modifying $k$ from three to four.

There are a few other related works. In [14], the authors discuss how to help users to quantify their preferences as a set of weightings. Their solution is based on presenting users a set of objects to choose, and try to infer the users’ weightings based on the objects that they have chosen. In the why-not paradigm, users are quite clear with which are the missing objects and our job is to explain to them why those objects are missing. In [15], the notion of reverse top-k queries is proposed. A reverse top-k query takes as input a top-k query, a missing object $m$, and a set of candidate weightings $W$. The output is a weighting $\bar{w} \in W$ that makes $m$ in its top-k result. Two solutions are given in [15]. The first one insists users to provide $W$ as input, which slightly limits its practicability. The second one does not require users to provide $W$, however, it only works when the top-k queries involve two attributes. Although the problems look similar, answering why-not questions on top-k SQL queries indeed does not require users to provide $W$.

In [10], the notion of why-not questions on reverse skyline queries is proposed. Given a set of data points $D$ and a query $q$, the dynamic skyline of a data point $p \in D$ is the skyline of a transformed data space using $p$ as the origin and the reverse skyline [16] returns the objects whose dynamic skyline contains the query point $q$. The why-not question to a reverse skyline query then asks about why a specific data point $p \in D$ is not in the reverse skyline of $q$. In [10], explanations using both data-refinement and query-refinement are discussed. In the data-refinement approach, the explanation is based on modifying the values (in minimal) of $p \in D$ such that $p$ appears in the reverse skyline of $q$. In the query-refinement approach, the explanation is based on modifying the query point $q$ until $p$ is in its reverse skyline. In addition to that, the discussion of why-not questions has also been extended to the contexts of spatial keyword top-k queries [17], reverse top-k queries [18], and graph matching [19]. All these works above, however, are not SQL oriented.

<table>
<thead>
<tr>
<th>ID</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>240</td>
<td>60</td>
</tr>
<tr>
<td>P2</td>
<td>233</td>
<td>60</td>
</tr>
<tr>
<td>P3</td>
<td>340</td>
<td>70</td>
</tr>
<tr>
<td>P4</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>P5</td>
<td>140</td>
<td>100</td>
</tr>
<tr>
<td>P6</td>
<td>150</td>
<td>50</td>
</tr>
</tbody>
</table>

(a) An example table $U$

<table>
<thead>
<tr>
<th>ID</th>
<th>0.5<em>A+0.5</em>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>P3</td>
<td>205</td>
</tr>
<tr>
<td>P1</td>
<td>150</td>
</tr>
<tr>
<td>P2</td>
<td>147.5</td>
</tr>
<tr>
<td>P5</td>
<td>120</td>
</tr>
<tr>
<td>P6</td>
<td>100</td>
</tr>
<tr>
<td>P4</td>
<td>85</td>
</tr>
</tbody>
</table>

(b) Ranking under original weightings

Fig. 1. Motivating example.
3 Answering Why-Not Questions on Top-k SPJ Queries

In this section, we first focus on answering why-not questions on top-k SQL queries with SPJ clauses. We will extend the discussion to why-not top-k SQL queries with GROUP BY and aggregation in the next section. For easy reading, Table 1 summarizes the most frequently used symbols in this paper.

### 3.1 The Problem and The Explanation Model

We consider a top-k SPJ query \( Q \) with a set of Select-Project-Join clauses \( \text{SPJ} \), a monotonic scoring function \( f \) and a weighting vector \( \bar{w} = [w[1], w[2], \ldots, w[d]] \), where \( d \) is the number of attributes in the scoring function. For simplicity, we assume a larger value means a better score (and rank) and the weighting space subject to the constraint \( \sum w[i] = 1 \) where \( 0 \leq w[i] \leq 1 \). We only consider conjunctions of predicates \( P_1 \land \ldots \land P_n \), where each \( P_i \) is either a selection predicate \( "A_i \ op \ v_i" \) or a join predicate \( "A_i \ op A_j" \), where \( A \) is an attribute, \( v \) is a constant, and \( op \) is a comparison operator.

For simplicity, our discussion focuses on \( \geq \) comparison because generalizing our discussion to other comparison operators is straightforward. The query result would then be a set of \( k \) tuples whose scores are the largest (in case tuples with the same scores are tied at rank \( k \)), only one of them is returned.

Initially, a user issues an original top-k SPJ query \( Q_o(SPJ_{i1}, k_o, \bar{w}_o) \) on a dataset \( D \). After she gets the query result, denoted as \( R_o \), she may pose a why-not question with a set of missing tuples \( Y = \{y_1, \ldots, y_l\} (l \geq 1) \), where \( y_i \) has the same set of projection attributes as \( Q_o \). In this paper, we adopt the query-refinement approach in [8] so that the system returns the user a refined query \( Q' = (SPJ'_{i1}, k', \bar{w}') \), whose result \( R' \) includes \( Y \) and \( R_o \) i.e., \( \{Y \cup R_o\} \subseteq R' \). It is possible that there are indeed no refined queries \( Q' \) that can include \( Y \) (e.g., \( Y \) contains a missing tuple whose expected attribute values indeed do not exist in the database). For those cases, the system will report to the user about her error.

There are possibly multiple refined queries for being the answers to a why-not question \( (Q_o, Y) \). We thus use \( \Delta_{SPJ} \), \( \Delta_k \) and \( \Delta_w \) to measure the quality of a refined query \( Q' \), where \( \Delta_k = k' - k_o \), \( \Delta_w = ||\bar{w}' - \bar{w}_o||_2 \), and \( \Delta_{SPJ} \) is defined based on four different types of edit operations of SPJ clauses adopted in [8]:

- \((c_1)\) modifying the constant value of a selection predicate in the \text{WHERE} clause.
- \((c_2)\) adding a selection predicate in the \text{WHERE} clause.
- \((c_3)\) adding/removing a join predicate in the \text{WHERE} clause.
- \((c_4)\) adding/removing a relation in the \text{FROM} clause.

Following [8], we do not allow other edit operations such as changing the projection attributes (because users usually have a clear intent about the projection attributes). Note that there is no explicit edit operation for removing a selection predicate, since it is equivalent to modifying the constant value in the predicate to cover the whole domain of the attribute. Furthermore, we also do not consider modifying the joins to include self-join. Let \( c_i \) denote the cost of the edit operation \( e_i \) and we follow [8] to set \( c_1 = 1, c_2 = 3, c_3 = 5, c_4 = 7 \). So, \( \Delta_{SPJ} = \sum_{1 \leq i \leq 4} (c_i \times n_i) \), where \( n_i \) is the number of edit operations \( e_i \) used to obtain the refined query \( Q' \). In order to capture a user’s tolerance to the changes of SPJ clauses, \( k \), and \( \bar{w} \) on her original query \( Q_o \), we first define a basic penalty model that sets the penalties \( \lambda_{SPJ}, \lambda_k \) and \( \lambda_w \) to \( \Delta_{SPJ}, \Delta_k \) and \( \Delta_w \), respectively, where \( \lambda_{SPJ} + \lambda_k + \lambda_w = 1 \):

\[
\text{Basic Penalty} = \lambda_{SPJ} \Delta_{SPJ} + \lambda_k \Delta_k + \lambda_w \Delta_w.
\]
Note that the basic penalty model is able to capture both the similar and precise requirements. Specifically, a refined query \( Q' \) that minimizes Basic Penalty implies it is similar to the original query \( Q_o \). To make the result precise (i.e., having fewer extra tuples), we can set a larger penalty \( \lambda_k \) to \( \Delta k \) such that modifying \( k \) significantly is undesired.

The basic penalty model, however, has a drawback because \( \Delta k \) generally could be a large integer (as large as \( |D| \)) whereas \( \Delta w \) and \( \Delta SPJ \) are generally smaller. One possible way to mitigate this discrimination is to normalize them respectively.

**[Normalizing \( \Delta SPJ \)]** We normalize \( \Delta SPJ \) using the maximum editing cost \( \Delta SPJ_{max} \).

**Definition 1 (maximum editing cost \( \Delta SPJ_{max} \)).** Given the original query \( Q_o \), the maximum editing cost \( \Delta SPJ_{max} \) is the editing cost of obtaining a refined SPJ query \( Q_{SPJ}^{max} \) whose (1) SPJ constructs most deviated from the SPJ constructs of the original query \( Q_o \) (based on the four types of edit operations \( e_1 \) to \( e_4 \)) and (2) with a query result that includes all missing tuples \( Y \) and the original query result \( R_o \).

**Example 1 (maximum editing cost \( \Delta SPJ_{max} \)).** Fig. 2 shows an example database \( D \) with three base tables \( T_1 \), \( T_2 \), and \( T_3 \). Assume a user has issued the following top-2 SQL query:

\[
Q_o:
SELECT B
FROM T_1, T_2
WHERE T_1.A = T_2.A AND D \geq 400
ORDER BY 0.5 * D + 0.5 * E
LIMIT 2
\]

By referring to Fig. 3 (the join result of \( T_1 \bowtie T_2 \)), the result \( R_o \) of the top-2 query is: [Gary, Alice]. Assuming the missing tuples set \( Y \) of the why-not question is [Chandler]. Then, \( Q_{SPJ}^{max} \) is:

\[
Q_{SPJ}^{max}
SELECT B
FROM T_1, T_2, T_3
WHERE T_1.A = T_2.A AND T_1.A = T_3.A
AND C \geq 50
AND D \geq 100
AND E \geq 50
AND F \geq 60
AND G \geq 200
AND H \geq 60
\]

1. To obtain \( Q_{SPJ}^{max} \) in this example, we (i) add all other tables (e.g., \( T_3 \)) with the corresponding join conditions, and (ii) add/refine selection predicates for all attributes so that \( Y \cup R_o \) is included the query result.

Accordingly, \( \Delta SPJ_{max} = 1 \times c_1 + 5 \times c_2 + 1 \times c_3 + 1 \times c_4 = 1 + 5 \times 3 + 3 + 7 = 28 \).

**[Normalizing \( \Delta k \)]** We normalize \( \Delta k \) using \( (r_o - k_o) \), where \( r_o \) is the worst rank with minimal edits.

**Definition 2 (worst rank with minimal edits).** The worst rank with minimal edits \( r_o \) is the worst rank among all tuples in \( Y \cup R_o \) of a refined top-\( k \) SQL query \( Q_{SPJ}^{min} \) whose (1) SPJ constructs least deviated from the original query \( Q_o \) (measured by \( c_1 \) to \( c_4 \)), (2) using the original weighting \( w_o \), (3) with a query result that includes all missing tuples \( Y \) and the original query result \( R_o \), and (4) the modification of \( k \) is minimal.

To explain why \( r_o \) is a suitable value to normalize \( \Delta k \), we first remark that we could normalize \( \Delta k \) using the cardinality of the join result of \( Q_o \), because that is the worst possible rank. But to get a more reasonable normalizing constant, we look at Equation (1). First, to obtain the “worst” but reasonable value of \( \Delta k \), we can assume that we do not modify the weighting, leading to condition (2) in Definition 2. Similarly, we do not want to modify the SPJ constructs so much but we hope the SPJ constructs at least do not filter out the missing tuples \( Y \) and the original query result \( R_o \), leading to conditions (1) and (3) in Definition 2. So, based on Example 1, \( Q_{SPJ}^{min} \) is:

\[
Q_{SPJ}^{min}
SELECT B
FROM T_1, T_2
WHERE T_1.A = T_2.A AND D \geq 200
ORDER BY 0.5 * D + 0.5 * E
LIMIT 7
\]

We note that the following is not \( Q_{SPJ}^{min} \) although it also satisfies conditions (1) to (3) because its modification of \( k \) is from two to eight, which is not minimal (condition 4) comparing with the true \( Q_{SPJ}^{min} \) above:

\[
Q_{SPJ}^{min}'
SELECT B
FROM T_1, T_2
WHERE T_1.A = T_2.A AND D \geq 100
ORDER BY 0.5 * D + 0.5 * E
LIMIT 8
\]

So, based on \( Q_{SPJ}^{min} \), \( \Delta k \) will be normalized by \( (r_o - k_o) = (7 - 2) = 5 \).

**[Normalizing \( \Delta w \)]** Let the original weighting vector \( w_o = [w_o[1], w_o[2], \ldots, w_o[d]] \), we normalize \( \Delta w \) using \( \sqrt{1 + \sum |w_o[i]|^2} \), because:

**Lemma 1.** In our concerned weighting space, given \( \tilde{w}_o \) and an arbitrary weighting vector \( \tilde{w} = [w[1], w[2], \ldots, w[d]] \), \( \Delta w \leq \sqrt{1 + \sum |w[i]|^2} \).
Table 2 lists some examples of refined queries that could be used to improve the efficiency of our algorithm.

### 3.2 Problem Analysis

Answering a why-not question is essentially searching for the best refined SPJ clauses and weighting in (1) the space $S_{SPJ}$ of all possible refined SPJ clauses and in (2) the space $S_{w}$ of all possible weightings, respectively. It is not necessary to search for $k$ because once the best set of SPJ clauses and the best weighting $\bar{w}$ are found, the value of $k$ can be accordingly set as the worst rank of tuples in $Y \cup R_o$. The search space $S_{SPJ}$ can be further divided into two: (1a) the space of the query schemas $S_{QS}$ and (1b) the space of all the selection conditions $S_{sel}$. A query schema $QS$ represents the set of relations in the FROM clause and the set of join predicates in the WHERE clause. A selection condition represents the set of selection predicates in the WHERE clause.

First, the space $S_{QS}$ is $O(2^n)$, where $n$ is the number of relations in the database. Second, the space $S_{sel}$ given a query schema is $O(\prod_{i=1}^{m}(|A_i| + 1))$, where $m$ is the number of attributes in that query schema, $|A_i|$ is the number of distinct values in attribute $A_i$, and $|A_i| + 1$ takes into account attribute $A_i$ can optionally be or not be added to the selection condition. Third, the space $S_w$ is infinite. Hence, it is obvious that finding the exact best refined query from $S_{QS} \times S_{sel} \times S_w$ is impractical.

### 3.3 The Solution

According to the problem analysis presented above, finding the best refined query is computationally difficult. Therefore, we trade the quality of the answer with the running time.

Let us start the discussion by illustrating our basic idea under the assumption that there is only one missing tuple $y$. First, we observe that not every query schema can generate a query whose results contain $\{y\} \cup R_o$. For Example 1, if the missing tuple $y$ is [Henry], then the query schema $T_1 \Join T_2$ does not include $y$. In this case, no matter how we exhaust the search space $S_{sel}$ and $S_w$, we cannot generate a valid refined query. In other words, during the search for $y = [Henry]$, if we are able to filter out query schema $T_1 \Join T_2$, then the subsequent search in $S_{sel}$ and $S_w$ can be skipped accordingly. Hence, we enumerate the space $S_{QS}$ prior to $S_{sel}$ and $S_w$. Similarly, we enumerate $S_{sel}$ prior to $S_w$ since some predicates in $S_{sel}$ may filter out the tuples in $\{y\} \cup R_o$.

When enumerating $S_{QS}$, there could be multiple query schemas that satisfy the requirement of generating a query whose results contain $\{y\} \cup R_o$. Our idea here is similar to [8], which starts from the original query schema $QS_w$ carries out incremental modification to $QS_s$ (using edit operation $e_i$), and stops once we have found a query schema $QS'$ for which queries based on that can include $\{y\} \cup R_o$ in the result. If no such a query schema is found, we report to the user that no refined query can answer her why-not question.
Once the target query schema $QS'$ is found, we next enumerate all possible selection conditions $S_{sel}$ that can be derived from $QS'$ with a set $S_w$ of weighting vectors. The set $S_w$ includes a random sample $S$ of vectors $\mathbf{\tilde{w}}_1, \mathbf{\tilde{w}}_2, \ldots, \mathbf{\tilde{w}}_s$ from the weighting space $S$ and the original weighting $\mathbf{w}_r$. That is, $S_w = \{\mathbf{w}_r\} \cup S$. For each selection condition $sel_i \in S_{sel}^{QS'}$ and each weighting $\mathbf{\tilde{w}}_j \in S_w$, we formulate a refined top-k SQL query $Q_{ij}$ and execute it using a progressive top-k SQL algorithm (e.g., [26], [27], [28]), which progressively reports each top rank tuple one-by-one, until all tuples in $\{y\} \cup R_o$ come forth to the result set at ranking $r_{ij}$. So, after $S_{sel}^{QS'} \cdot |S_w|$ progressive top-k SQL executions, we have $S_{sel}^{QS'} \cdot |S_w|$ refined queries. Finally, using (1) $r_{ij}$ as the refined $k'$, (2) $sel_i$, and $QS'$ to formulate the refined SPJ clauses $SPJ'$, and (3) $\mathbf{\tilde{w}}_j$ as the refined weighting $\mathbf{w}'$, the refined query $Q'(SPJ', k', \mathbf{w}')$ with the least penalty is returned to the user as the why-not answer.

The basic idea above incurs many progressive top-k SQL executions. In the following, we present our algorithm in detail. The algorithm consists of three phases and includes different optimization techniques in order to reduce the execution time.

**[PHASE-1]** In this phase, we start from the original query schema $QS$, and do incremental modification to $QS$ (using edit operation $e_1$) and stop until we find a query schema $QS'$ for which queries based on that can include $\{y\} \cup R_o$ in the result. As all the subsequent considered refined queries will be based on $QS'$, we materialize the join result $J$ in memory in order to avoid repeated computation of the same join result based on $QS'$ in the subsequent phase. If no such a query schema is found, we report to the user that no refined query can answer her why-not question. Finally, we randomly sample $s$ weightings $\mathbf{\tilde{w}}_1, \mathbf{\tilde{w}}_2, \ldots, \mathbf{\tilde{w}}_s$ from the weighting space and add them into $S_w$ in addition to $\mathbf{w}_r$.

**[PHASE-2]** Next, for a subset $S_{sel}^{QS} \subseteq S_{sel}^{QS'}$ of selection conditions that can be derived from $QS'$ and weighting vectors $\mathbf{\tilde{w}}_j \in S_w$, we execute a progressive top-k SQL query on the materialized join result $J$ using a selection condition $sel_i \in S_{sel}^{QS'}$ and a weighting $\mathbf{\tilde{w}}_j \in S_w$ until a stopping condition is met. From now on, we denote a progressive top-k SQL execution as:

$$r_{ij} = \text{TOPK}(sel_i, \mathbf{\tilde{w}}_j, \text{STOPPING-CONDITION}),$$

where $r_{ij}$ denotes the rank when all $\{y\} \cup R_o$ come forth to the results under selection condition $sel_i$ and weighting $\mathbf{\tilde{w}}_j$.

In the basic idea mentioned above, we have to execute $|S_{sel}^{QS'}| \cdot |S_w|$ progressive top-k SQL executions. In Section 3.3.1, we will illustrate that we can just focus on a much smaller subset $S_{sel}^{QS} \subseteq S_{sel}^{QS'}$ without jeopardizing the quality of the answer. Consequently, the number of progressive top-k SQL executions could be largely reduced to $|S_{sel}^{QS'}| \cdot |S_w|$.

Furthermore, the original stopping condition in our basic idea is to proceed the topk execution on $J$ until all tuples in $\{y\} \cup R_o$ come forth to the result. However, if some tuples in $\{y\} \cup R_o$ rank very poorly under some weighting $\mathbf{\tilde{w}}_j$, the corresponding progressive top-k SQL operation may be quite slow because it has to access many tuples in the materialized join result $J$. In Section 3.3.2, we present a much more aggressive and effective stopping condition that makes most of those operations stop early before $\{y\} \cup R_o$ come forth to the result.

Finally, we present a technique in Section 3.3.3 that can identify some weightings in $S_w$ whose generated refined queries have poorer quality, thereby skipping the topk execution for those weightings to gain better efficiency.

**[PHASE-3]** Using $r_{ij}$ as the refined $k'$, $sel_i$ and $QS'$ as the refined SPJ clauses $SPJ'$, and $\mathbf{\tilde{w}}_j$ as the refined weighting $\mathbf{w}'$, the refined query $Q'(SPJ', k', \mathbf{w}')$ with the least penalty is returned to the user as the why-not answer.

### 3.3.1 Excluding Selection Predicates That Could not Yield Good Refined Queries

In the basic idea, we have to enumerate all possible selection conditions based on the query schema $QS'$ chosen in PHASE-1. As mentioned, if there are $m$ attributes in $QS'$ and $|A|$ is the number of distinct values in attribute $A_i$ in that query schema, there would be $O(|A|^m ((|A| + 1) \cdot m)$ possible selection conditions. However, the following lemma can help us to exclude selection conditions that could not yield good refined queries:

**Lemma 2.** If we need to modify the original selection condition by modifying the constant value $v_i$ of its selection predicate $P$ in the form of $A_i \geq v_i$ (edit operation $e_1$), or adding a selection predicate $P$ in the form of $A_i \geq v_i$ (edit operation $e_2$) to the selection condition, we can simply consider only one possibility of $P$, which is $A_i \geq v_{i min}$, where $v_{i min}$ is the minimum value of attribute $A_i$, among tuples in $\{y\} \cup R_o$, because $P$ in any other form would not lead to better refined top-k SQL queries whose penalties are better than $P$ as $A_i \geq v_{i min}$.

**Proof.** Let $sel = \{A_1 \geq v_1, \ldots, A_m \geq v_l\}$ ($l \leq m$) be the selection condition after modification and particularly $A_i \geq v_i$ be a predicate $P'$ in $sel'$ which gets modified/added from the original selection condition.

First, $v'_i$ in $P'$ has to be smaller than or equal to $v_{i min}$ or otherwise some tuples in $\{y\} \cup R_o$ would get filtered away.

Second, comparing a predicate $P$: $A_i \geq x$, with $x$ as any value smaller than or equal to $v_{i min}$, and the predicate $P_{min}$: $A_i \geq v_{i min}$, these two predicates incur the same $\Delta SPJ$ to the original selection condition.

Third, as the predicate $P$: $A_i \geq x$, is less restrictive than the predicate $P_{min}$: $A_i \geq v_{i min}$, more tuples could pass $P$. So, given a weighting $\mathbf{\tilde{w}}$, the worst rank of

3. Let $A$ denote all possible subsets of attributes in $QS'$. For each subset $A_0 \in A$, each enumerated selection condition is in the form of \( \land (A_i \geq v_j) \), $\forall A_i \in A_0$; $v_j$ is enumerated from the domain of $A_i$. Only selection conditions that do not filter $\{y\} \cup R_o$ are added to $S_{sel}^{QS'}$.

4. There could be multiple tuples in $J$ that match $y$. One option is to randomly choose one in $J$ that matches $y$ as the missing tuple. Another option is to repeat the whole process for each matching tuple and regard the one with the lowest penalty as the answer. It is a tradeoff between efficiency and quality. In this paper, we adopt the first option.
\{y\} \cup R_o under P cannot be better than that under \(P_{\text{min}}\). That implies \(\Delta k\) under P would not be better than that under \(P_{\text{min}}\) as well. Therefore, we conclude that top-k SQL queries based on P would not lead to better refined top-k SQL queries based on \(P_{\text{min}}\). \(\square\)

Consider Example 1 again. The query schema \(QS^t\) that can include back the missing tuple (Chandler) would lead to a join result like Fig. 3. Originally, we have to consider eight predicates when dealing with attribute \(D\), which are \(D \geq 500\), \(D \geq 400\), \(D \geq 300\), \(D \geq 290\), \(D \geq 280\), \(D \geq 250\), \(D \geq 210\), and \(D \geq 100\). By using Lemma 2, we just need to consider \(D \geq 200\) because among \(\{y\} \cup R_o = \{\text{Chandler, Gary, Alice}\}\), their attribute values of \(D\) are 200, 400, and 500, with 200 as the minimum. Similar for attribute \(E\), by using Lemma 2, we just need to consider \(E \geq 80\). The above discussion can be straightforwardly generalized to other comparisons including \(\leq\), \(<\), and \(>\).

### 3.3.2 Stopping a Progressive Top-k SQL Operation Earlier

In PHASE-2 of our algorithm, the basic idea is to execute the progressive top-k SQL query until \(\{y\} \cup R_o\) come forth to the results, with rank \(r_{\text{y}}\). In the following, we show that it is actually possible for a progressive top-k SQL execution to stop early even before \(\{y\} \cup R_o\) come forth to the result.

Consider an example where a user specifies a top-k SQL query \(Q_{o,2}(SPJ_{o,2}, k_o = 2, \vec{w}_o)\) on a data set \(D\) and poses a why-not question about a missing tuple \(y\). Assume that the list of weightings \(S_w = \{\vec{w}_1, \vec{w}_2, \vec{w}_3\}\) and \(\text{topk}(\vec{w}_1, \vec{w}_2)\), \(\text{until-see}(\{y\} \cup R_o)\) is first executed, where all tuples in \(\{y\} \cup R_o\) come forth to the result at rank 7. Now, we have our first candidate refined query \(Q'_{o,2}(sel_1, r_{10} = 7, \vec{w}_o)\) with \(\Delta k = 7 - 2 = 5\) and \(\Delta w = |w_o - w_o| = 0\). The corresponding penalty, denoted as, \(\text{Pen}_{Q'_{o,2}}\), could be calculated using Equation (2). Remember that we want to find the refined query with the least penalty \(\text{Pen}_{\text{min}}\). So, at this moment, we set \(\text{Pen}_{\text{min}} = \text{Pen}_{Q'_{o,2}}\).

According to our basic idea outlined above, we should next execute another progressive top-k SQL using the selection condition \(sel_1\) and another weighting vector, say, \(\vec{w}_1\), until \(\{y\} \cup R_o\) come forth to the result with a rank \(r_{11}\). However, we notice that the skyline property in the answer space can help to stop that operation earlier, even before \(\{y\} \cup R_o\) are seen. Given the first candidate refined query \(Q'_{o,2}(sel_1, r_{10} = 7, \vec{w}_o)\) with \(\Delta w = 0\) and \(\Delta k = 5\), any other candidate refined queries \(Q'_{o,1}\) with \(\Delta k > 5\) must be dominated by \(Q'_{o,2}\). In our example, after the first executed progressive top-k SQL execution, all the subsequent progressive top-k SQL executions with \(sel_1\) can stop once \(\{y\} \cup R_o\) do not show up in the top-7 tuples.

Fig. 5 illustrates the answer space of the example. The idea above essentially means a progressive top-k SQL execution can stop if \(\{y\} \cup R_o\) do not show up in result after returning the top-7 tuples (i.e., \(\Delta k > 5\); see the dotted region). For example, consider \(Q'_{o,1}\) in Fig. 5 whose weighting is \(w_1\) and \(\{y\} \cup R_o\) show up in the result only when \(k = 9\), i.e., \(\Delta k = 9 - 2 = 7\). So, the progressive top-k SQL associated with \(Q'_{o,1}\) can stop after \(k = 7\), because after that \(Q'_{o,1}\) has no chance to dominate \(Q'_{o,2}\) anymore. In other words, the progressive top-k SQL operation associated with \(Q'_{o,1}\) does not wait to reach \(k = 9\) where \(\{y\} \cup R_o\) come forth to the result but stop early when \(k = 7\).

While useful, we can actually be even more aggressive in many cases. Consider another candidate refined query, say, \(Q'_{o,2}\), in Fig. 5. Assume that \(r_{12} = \text{topk}(\{y\} \cup R_o)\), \(\text{until-see}(\{y\} \cup R_o)\) = 6 (i.e., \(\Delta k = 6 - 2 = 4\)), which is not covered by the above technique (since \(\Delta k \notin 5\)). However, \(Q'_{o,2}\) can also stop early, as follows. In Fig. 5, we show the normalized penalty Equation (2) as a slope \(\text{Pen}_{\text{min}} = \text{Pen}_{Q'_{o,2}}\) that passes through the best refined query so far (currently \(Q'_{o,2}\)). All refined queries that lie on the slope have the same penalty value as \(\text{Pen}_{\text{min}}\). After this, all refined queries that lie above the slope actually have a penalty larger than \(\text{Pen}_{\text{min}}\) and thus are inferior to \(Q'_{o,2}\). Therefore, similar to the skyline discussion above, we can determine an even tighter threshold ranking \(t^T\) for stopping the subsequent progressive top-k SQL operations with \(sel_1\):

\[
r^T = \Delta k^T + k_o, \text{ where } \Delta k^T = \{\left[\frac{\text{Pen}_{\text{min}} - \lambda_{\text{spj}} \Delta SPJ}{\Delta SPJ_{\text{max}} - \lambda_{\text{spj}} \Delta w \sqrt{1 + \sum w_i^2}}\right] r_o - k_o \}
\]

Equation (3) is a rearrangement of Equation (2) (with \(\text{Penalty} = \text{Pen}_{\text{min}}\)). Back to our example in Fig. 5, given that the weighting of candidate refined query \(Q'_{o,2}\) is \(\vec{w}_2\), we can first compute its \(\Delta w_2\) value. Then, we can project \(\Delta w_2\) onto the slope \(\text{Pen}_{\text{min}}\) (currently \(\text{Pen}_{\text{min}} = P_{Q'_{o,2}}\)) to obtain the corresponding \(\Delta k^T\) value, which is three in Fig. 5. That means, if we carry out a progressive top-k SQL operation using \(sel_1\) as the predicates and \(\vec{w}_2\) as the weighting, and if \(\{y\} \cup R_o\) still do not appear in result after the top-5 tuples (\(r^T = \Delta k^T + k_o = 3 + 2 = 5\) are seen), then we can stop it early because the penalty of \(Q'_{o,2}\) is worse than the penalty \(\text{Pen}_{\text{min}}\) of the best refined query \(Q'_{o,2}\) seen so far.

Following the discussion above, we now have two early stopping conditions for the progressive top-k SQL algorithm: \(\text{until-see}(\{y\} \cup R_o)\) and \(\text{until-rank}-r^T\). Except
for the first progressive top-k SQL operation which \textsc{topk}(\text{sel}_1, \bar{w}_o, \text{until-see}-\{{y} \cup R_o\}) must be used, the subsequent progressive top-k SQL operations with \text{sel}_1 can use "\text{until-see}-\{{y} \cup R_o\} or \text{until-rank-r}^T" as the stopping condition. We remark that the conditions \text{until-rank-r}^T and \text{until-see}-\{{y} \cup R_o\} are both useful. For example, assume that the actual worst rank of \{{y} \cup R_o\} under \text{sel}_1 and \bar{w}_2 is four, which gives it a \Delta k = 2 (see $Q''_{12}$ in Fig. 5).

Recall that by projecting \Delta w_2 onto the slope of $Pen_{min} = Pen_{Q''_{12}}$, we can stop the progressive top-k SQL operation after $r^T = 3 + 2 = 5$ tuples have been seen. However, using the condition \text{until-rank-r}^T and \text{until-see}-\{{y} \cup R_o\}, we can stop the progressive top-k SQL operation when all \{{y} \cup R_o\} show up at rank four. This drives us to use "\text{until-see}-\{{y} \cup R_o\} or \text{until-rank-r}^T" as the stopping condition.

Finally, we remark that the optimization power of this technique increases while the algorithm proceeds. For example, after $Q''_{12}$ has been executed, the best refined query seen so far should be updated as $Q''_{12}$ (because its penalty is better than that of $Q_{12}$). Therefore, $Pen_{min}$ now is updated as $Pen_{Q''_{12}}$ and the slope $Pen_{min}$ should be updated to pass through $Q''_{12}$ now (the dotted slope in Fig. 5). Because $Pen_{min}$ is continuously decreasing, $\Delta k^T$ and thus the threshold ranking $r^T$ would get smaller and smaller and the subsequent progressive top-k SQL operations can terminate even earlier while the algorithm proceeds.

The above early stopping technique can be applied to the subsequent selection conditions. In our example, after selection condition \text{sel}_1 and turning to consider selection condition \text{sel}_2 with the set of weightings $S_w$, we can derive $r^T$ based on equation (3) by simply reusing the $Pen_{min}$ obtained from \text{sel}_1.

### 3.3.3 Skipping Progressive Top-k SQL Operations

In PHASE-2 of our algorithm, the basic idea is to execute progressive top-k SQL queries for all selection conditions in $S_{Q''_{12}}$ and all weightings in $S_w$. After the discussion in Section 3.3.2, we know them some progressive top-k SQL executions can stop early. We now illustrate three pruning opportunities where some of those executions could be skipped entirely.

The first pruning opportunity is based on the observation from [15] that under the same selection condition \text{sel}_1, similar weighting vectors may lead to top-k SQL results with more common tuples. Therefore, if an operation $\text{topk}(\text{sel}_1, \bar{w}_j, \text{until-see}-\{{y} \cup R_o\})$ for $\bar{w}_j$ has already been executed, and if a weighting $\bar{w}_i$ is similar to $\bar{w}_j$, then we can use the query result $R_{ij}$ of $\text{topk}(\text{sel}_1, \bar{w}_j, \text{until-see}-\{{y} \cup R_o\})$ to deduce the smallest $k$ value for $\text{sel}_1$ and $\bar{w}_i$. Let $k'$ be the deduced $k$ value for $\text{sel}_1$ and $\bar{w}_i$. If the deducted $k'$ is larger than the threshold ranking $r^T$, then we can skip the entire $\text{topk}(\text{sel}_1, \bar{w}_i, \text{stopping-condition})$ operation.

We illustrate the above by reusing our running example. Assume that we have cached the result sets of executed progressive top-k SQL queries. Let $R_{10}$ be the result set of the first executed query $\text{topk}(\text{sel}_1, \bar{w}_o, \text{until-see}-\{{y} \cup R_o\})$ and $R_o = \{t_5, t_6\}$. Assume that $R_{10} = \{t_1, t_2, t_3, t_4, t_5, t_6, y\}$. Then, when we are considering the next weighting vector, say, $\bar{w}_1$, in $S_w$, we first follow Equation (3) to calculate the threshold ranking $r^T$. In Fig. 5, projecting $\bar{w}_1$ onto slope $P_{Q''_{10}}$, we get $r^T = 4 + 2 = 6$. Next, we calculate the scores of all tuples in $R_{10}$ using $\bar{w}_1$ as the weighting. More specifically, let us denote the tuple in $\{y\} \cup R_o$ under weighting vector $\bar{w}_1$ as $\bar{t}_1$ if it has the worst rank among $\{y\} \cup R_o$. In the example, assume under $\bar{w}_1$, the scores of $t_1, t_2, t_3$ and $t_4$ are still better than $\bar{t}_1$, then the $k'$ value for $\text{sel}_1$ and $\bar{w}_1$ is at least $4 + 3 = 7$. Since $k'$ is worse than $r^T = 6$, we can skip the entire $\text{topk}(\text{sel}_1, \bar{w}_1, \text{stopping-condition})$ operation.

The above caching technique is shown to be the most effective between similar weighting vectors [15]. Therefore, we design the algorithm in a way that the list of weightings $S_w$ is sorted according to their corresponding $\Delta w$ values (of course, $\bar{w}_i$ is in the head of the list since $\Delta w_0 = 0$). In addition, the technique is general so that the cached result for a specific selection condition $\text{sel}_1$ can also be used to derive the smallest $k'$ value for another selection condition $\text{sel}_2$. As long as $\text{sel}_1$ and $\text{sel}_2$ are similar, the chance that we can deduce $k'$ from the cached result that leads to $\text{topk}$ operation pruning is also higher. So, we design the algorithm in a way that $\text{sel}_1$ is enumerated in increasing order of $\Delta w$ as well.

The second pruning opportunity is to exploit the best possible ranking of $\{y\} \cup R_o$ (under all possible weightings) to set up an early termination condition for some weightings, so that after a certain number of progressive top-k SQL operations have been executed under $\text{sel}_1$, operations associated with some other weightings for the same $\text{sel}_1$ can be skipped.

Recall that the best possible ranking of $\{y\} \cup R_o$ is $k_o + 1$, since $|\{y\} \cup R_o| = k_o + 1$. Therefore, the lower bound of $\Delta k$, denoted as $\Delta k_l$, equals 1. So, this time, we project $\Delta k_l$ onto slope $Pen_{min}$ in order to determine the corresponding maximum feasible $\Delta w$ value. We name that value as $\Delta w_l^f$. For any $\Delta w > \Delta w_l^f$, it means \{"$y\} \cup R_o\) has $\Delta k < \Delta k_l$, which is impossible. As our algorithm is designed to examine weightings in their increasing order $\Delta w$ values, when a weighting $w_j \in S_w$ has $|w_j - w_o| > \Delta w_l^f$, $\text{topk}(\text{sel}_1, \bar{w}_j, \text{stopping-condition})$ and all subsequent progressive top-k SQL operations $\text{topk}(\text{sel}_1, \bar{w}_j, \text{stopping-condition})$ where $l > j + 1$ could be skipped.

Reuse Fig. 5 as an example. By projecting $\Delta k_L = 1$ onto the slope $Pen_{Q''_{12}}$, we could determine the corresponding $\Delta w_l^f$ value. So, when the algorithm finishes executing a progressive top-k SQL operation for weighting $\bar{w}_2$, the algorithm can skip all the remaining weightings and proceed to examine the next selection condition.

As a remark, we would like to point out that the pruning power of this technique also increases while the algorithm proceeds. For instance, in Fig. 5, if $Q''_{12}$ has been executed, slope $Pen_{min}$ is changed from slope $Pen_{Q''_{12}}$ to slope $Pen_{Q''_{10}}$. Projecting $\Delta k_L$ onto the new $Pen_{min}$ slope would result in a smaller $\Delta w_l^f$, which in turn increases the chance of eliminating more weightings.

The last pruning opportunity is to set up an early termination condition for the whole algorithm. In fact, since we are sorting $\text{sel}_1$ in their increasing order of $\Delta w$, as soon as we encounter a $\Delta SPJ > (Pen_{min} - \lambda D \Delta w_{min}) \Delta SPJ_{min}$, we can skip all subsequent progressive top-k SQL operations and
terminate the algorithm. This equation is a rearrangement of the following equation:

\[ Pen_{\text{min}} < \lambda_{\text{spj}} \frac{\Delta SPJ}{\Delta SPJ_{\text{max}}} + \lambda_k \frac{\Delta k}{r_o - k_o}. \]  

(4)

### 3.3.4 How Large Should Be the List of Weighting Vectors?

Given that there are an infinite number of points (weightings) in the weighting space, how many sample weightings should we put into \( S_w \) in order to obtain a good approximation of the answer?

Recall that more sample weightings in \( S_w \) will increase the number of progressive top-k SQL executions and thus the running time. Therefore, we hope \( S_w \) to be as small as possible while maintaining good approximation. We say a refined query is the best-\( T \) percent refined query if its penalty is smaller than \((1 - T\%)\) percent refined queries in the whole (infinite) answer space, and we hope the probability of getting at least one such refined query is larger than a certain threshold \( Pr \):

\[ 1 - (1 - T\%)^n \geq Pr. \]  

(5)

Equation (5) is general. The sample size \( s \) is independent of the data size but is controlled by two parameters: \( T \) percent and \( Pr \). Following our usual practice of not improving usability (i.e., why not queries) by increasing the users’ burden (e.g., specifying parameter values for \( T \) percent, and \( Pr \)), we make \( T \) percent, and \( Pr \) system parameters and let users override their values only when necessary.

The pseudo-code of the complete algorithm is presented in Algorithm 1. It is self-explanatory and mainly summarizes what we have discussed above, so we do not give it a full walkthrough here.

### 3.3.5 Multiple Missing Tuples

To handle multiple missing tuples \( Y = \{y_1, \ldots, y_l\} \), \( l \geq 1 \), we just need little modification to the algorithm. First of all, a refined query needs to ensure \( \{Y \cup R_o\} \) (instead of \( \{y\} \) under \( \bar{y}\)) come forth to the result. Correspondingly, the stopping condition for the progressive top-k SQL algorithm becomes “\( \text{UNTIL-SEE} - \{Y \cup R_o\} \) OR \( \text{UNTIL-RANK-}\( r \)\( T \)\( P \)”. Apart from that, when generating the candidate selection conditions, we need to consider the minimum attribute values for all tuples in \( Y \cup R_o \).

### 4 Answering Why-Not Questions on Top-k SPJA Queries

In this section, we extend the discussion to why-not top-k SQL queries with GROUP BY and aggregation.

#### 4.1 The Problem and The Explanation Model

Initially, a user issues an original top-k SPJA query \( Q_o(\text{SPJA} \alpha, k_o, \bar{w}_o) \) on a dataset \( D \). After she gets the result \( R_o \), she may pose a why-not question about a set of missing groups \( Y = \{y_1, \ldots, y_l\} \) \( (l \geq 1) \), where \( g_i \) has the same set of projection attributes as \( Q_o \). Then, the system returns the user a refined query \( Q'(\text{SPJA}', k', \bar{w}') \), whose result \( R' \) includes \( Y \) and \( R_o \), i.e., \( \{Y \cup R_o\} \subseteq R' \). If there are indeed no refined queries \( Q' \) that can include \( Y \) and \( R_o \), the system will report to the user about her error.

**Algorithm 1. Answering a Why-not Top-k SPJA Question**

**Input:**
- The dataset \( D \), original top-k SQL query \( Q_o(\text{SPJA} \alpha, k_o, \bar{w}_o) \) and its query result \( R_o \), the missing tuple \( y \), penalty settings \( \lambda_{\text{spj}}, \lambda_k, \lambda_w, T\% \), and \( Pr \); edit cost for SPJA clauses: \( c_1, c_2, c_3 \) and \( c_4 \)

**Output:**
- A refined query \( Q'(\text{SPJA}', k', \bar{w}') \)

**Phase 1:**
1. Obtain \( Q_S \) and \( J \) by doing incremental modification to the original query schema \( Q_S \).
2. if \( Q_S \) does not exist
3. return “cannot answer the why-not question”;
4. end if
5. Construct \( S_{\text{spja}} \) based on Section 3.3.1;
6. Sort \( S_{\text{spja}} \) according to their \( \Delta s \) value;
7. Determine \( s \) from \( T\% \) and \( Pr \) using Equation (5);
8. Sample \( s \) weightings from the weighting space, add them and \( \bar{w}\) into \( S_w \);
9. Sort \( S_w \) according to their \( \Delta w \) values;

**Phase 2:**
10. \( Pen_{\text{min}} \leftarrow \infty \);
11. \( \Delta w \leftarrow \infty \);
12. for all \( s \) \( \in S_{\text{spja}} \) do
13. if \( \Delta SPJ > (Pen_{\text{min}} - \lambda_k \frac{\Delta k}{r_o - k_o}) \frac{\Delta SPJ_{\text{max}}}{\lambda_{\text{spj}}} \)
14. break; //Section 3.3.3 — early algorithm termination
15. end if
16. for all \( \bar{w} \) \( \in S_w \) do
17. if \( \Delta w > \Delta w' \) then
18. break; //Technique 3.3.3 — skipping weightings
19. end if
20. Compute \( r' \) based on Equation (3);
21. if there exist \( r' = |\{y\} \cup R_o| + 1 \) objects in some \( R_{ij} \in R \) having scores better than the worst rank of tuples in \( \{y\} \cup R_o \) under \( \bar{w}\) then
22. continue; //Section 3.3.3 — use cached result to skip a progressive top-k SQL operation
23. end if
24. \( (R_{ij}, r_{ij}) \leftarrow \text{TOPK}(s, \bar{w}, \text{UNTIL-SEE}-\{y\} \cup R_o) \) OR \( \text{UNTIL-RANK-}r' \); //Section 3.3.2 — stopping a progressive top-k SQL operation earlier
25. Compute \( Pen_{ij} \) based on Equation (2);
26. Add \( R_{ij} \) to \( R \);
27. if \( Pen_{ij} < Pen_{\text{min}} \) then
28. \( Pen_{\text{min}} \leftarrow Pen_{ij} \);
29. end if
30. end for
31. end for
32. end for

**Phase 3:**
33. Return the best refined query \( Q'(\text{SPJA}', k', \bar{w}') \) whose penalty equals \( Pen_{\text{min}} \);

Following [8], we do not modify the aggregation function and the group-by attributes in the original query. For the SPJA clauses, we adopt the four edit operations in Section 3 whereas the penalty function is similar to Equation (2):
The calculation of \( \Delta SPJA \) is the same as that of \( \Delta SPJ \). The cost of \( \Delta SPJ_{\text{max}} \) refers to the editing cost of obtaining a refined SPJA query \( Q'_{\text{spja}} \), whose definition is the same as \( Q_{\text{spja}} \) except that it needs to include all missing groups \( Y \) and the original result groups \( R_w \).

The problem definition is as follows. Given a why-not question \( (Q_o, Y) \), where \( Y \) is a set of missing groups and \( Q_o \) is the user’s initial query with result \( R_o \), our goal is to find a refined top-k SQL query \( Q'(SPJA', k', \bar{w}') \) that includes \( Y \cup R_w \) in the result with the smallest penalty with respect to Equation (6). Again, we do not explicitly ask users to specify the values for \( \lambda_{\text{spja}}, \lambda_k \) and \( \lambda_w \), but prompt users to answer a simple multiple-choice question listed in Fig. 4.

### 4.2 Problem Analysis

Since we allow the same set of edit operations as top-k SPJ queries, the problem complexity of answering why-not top-k SPJA questions follows the one presented in Section 3.2.

### 4.3 The Solution

The following changes are required to extend Algorithm 1 to handle why-not top-k SPJA questions.

First, when materializing the join result \( J \) of \( QS' \) (Line 1), we sort \( J \) using the group-by attributes so as to facilitate the subsequent grouping step. Second, we do not apply Lemma 2. Recall that Lemma 2 was proposed to exclude selection conditions that could not yield good refined queries. Specifically, when enumerating selection conditions, Lemma 2 considers only selection predicates in the form of \( A_i \geq v_{i_{\text{min}}} \), where \( v_{i_{\text{min}}} \) is the minimum value of attribute \( A_i \) among tuples in \( \{y\} \cup R_w \). That works because a missing tuple \( y \) corresponds to a single value in attribute \( A_i \). However, in the context of missing groups, a missing group \( Y \) could correspond to multiple values in attribute \( A_i \). Therefore, Lemma 2 might miss some good refined queries.

**Example 2.** Consider the following top-k SPJA query based on the data set in Fig. 2:

\[
\begin{align*}
Q_o: & \quad \text{SELECT } B \\
& \quad \text{FROM } T_1, T_2 \\
& \quad \text{WHERE } T_1.A = T_2.A \text{ AND } D \geq 400 \\
& \quad \text{GROUP BY } B \\
& \quad \text{ORDER BY } \text{AVG}(0.5 * D + 0.5 * E) \\
& \quad \text{LIMIT } 2
\end{align*}
\]

The result \( R_o \) is \{Gary, Alice\}. Let us assume the why-not question is asking for the missing group \{Daniel\}. From Fig. 3, we see that the group \{Daniel\} is composed by two base tuples (i.e., the two rows with \( B \) equals Daniel), with a group score average as \((185 + 155)/2 = 170\). If we follow Lemma 2 (using the minimum attribute \( D \)'s values among Gary, Alice, and Daniel) to modify the selection condition \( D \geq 400 \) to \( D \geq 100 \), the top-4 would then become:

<table>
<thead>
<tr>
<th>GROUP BY B</th>
<th>AVG(0.5 * D + 0.5 * E)</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gary</td>
<td>300</td>
<td>1</td>
</tr>
<tr>
<td>Alice</td>
<td>240</td>
<td>2</td>
</tr>
<tr>
<td>Bob</td>
<td>175</td>
<td>3</td>
</tr>
<tr>
<td>Daniel</td>
<td>170</td>
<td>4</td>
</tr>
</tbody>
</table>

If we do not follow Lemma 2, we can modify the selection condition \( D \geq 400 \) to, say, \( D \geq 300 \). In this manner, the top-3 result already includes all required groups (Gary, Alice, Daniel) due to the fact that the group \{Bob\} has been excluded by \( D \geq 300 \). Thus, we obtain a better refined query by (i) improving \( \Delta k \) from 2 to 1 and (ii) keeping \( \Delta SPJ \) and \( \Delta w \) unchanged.

Lastly, when using the caching technique described in Section 3.3, we cannot just cache the resulting groups or otherwise we do not have the base tuples that contribute to each group to derive the new score using another weighting (or selection condition). Therefore, for that technique, we also cache the base tuples for each resulting group.

### 5 Experiments

We evaluate our proposed techniques through a case study and a performance study. By default, we set the system parameters \( T \) percent and \( Pr \) as 0.5 percent and 0.7, respectively, resulting in a sample size of 241 weighting vectors. The algorithms are implemented in C++ and the experiments are run on a Ubuntu PC with Intel 3.4 GHz i7 processor and 16 GB RAM. For comparisons, we implement three baseline algorithms:

- **Top-k**: The algorithm in our early work [9], which modifies only top-k (i.e., \( \bar{w} \) and \( k \)) but not SQL's constructs. This baseline may not be able to answer some why-not questions if the missing tuple is indeed filtered by some SQL constructs.
- **Top-k-adapted**: This baseline adapts our early work [9] as follows: when sampling \( s \) weightings, we draw samples from a restricted weighting space constituted by the missing tuples \( Y \) and the original result \( R_w \), as well as the data points that are incomparable with \( Y \cup R_w \). It was proved by us that such a restricted weighting space is smaller than the entire weighting space and therefore samples from that space could have higher quality [9]. Since the construction of the restricted weighting space depends on data points that are incomparable with \( Y \cup R_w \), and the selection condition \( sel_i \) would influence the number of such incomparable data points, so we have to construct a specific restricted weighting space for each \( sel_i \in S'_{\text{spja}} \).
- **Staged**: This algorithm employs a two-stage process. (1) Ignore the top-k part and first apply the why-not SPJA solution in [8] to get the refined query. (2) If the desired tuples still do not show up in the top-k, increase \( k \) accordingly.

### 5.1 Case Study

The case study is based on the NBA data set, whose schema is shown in Fig. 6. It contains statistics of all NBA players from 1,973–2,009 with four tables: (i) \( Player \) (4,051 tuples) records players’ name and their career start year, (ii) \( Career \) (4,051 tuples) stores players’ performance in their whole career, (iii) \( Regular \) (21,961 tuples) and (iv) \( Playoffs \) (8,341 tuples) contain players’ performance year-by-year in regular games and playoffs games, respectively. We designed four cases based on four queries on the data set. Then, we put the cases and the results returned by our approach, Top-k,
Top-k-adapted, and Staged online\(^5\) and recruited 100 qualified AMT (Amazon Mechanical Turk) workers to grade the user satisfaction about the why-not answers in a scale of 1 to 4 (1 - very dissatisfied, 2 - dissatisfied, 3 - satisfied, 4 - very satisfied). Table 3 shows a summary of the results in terms of execution time, penalty and user satisfaction. In the following, we present some more details.

Case 1 (Finding the top-3 players who have played at least 1,000 games in NBA history). The first case study was to find the top-3 players with at least 1,000-game experience in the NBA history. Therefore, we issued a query \(Q_1\):

\[
Q_1: \quad \text{SELECT P.Name}
\]
\[
\text{FROM Player P, Career C}
\]
\[
\text{WHERE P.PID = C.PID AND C.GP} \geq 1,000
\]
\[
\text{ORDER BY (} \frac{7}{5} \text{* C.PPG + } \frac{7}{8} \text{* C.RPG + } \frac{1}{2} \text{* C.APG}
\]
\[
+ \frac{1}{2} \text{* C.SPG} + \frac{1}{2} \text{* C.BPG} + \frac{1}{2} \text{* C.FG percent}
\]
\[
+ \frac{1}{2} \text{* C.FT percent) DESC}
\]
\[
\text{LIMIT 3}
\]

The initial result was\(^6\):

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>GP</th>
<th>PPG</th>
<th>RPG</th>
<th>APG</th>
<th>SPG</th>
<th>BPG</th>
<th>FG%</th>
<th>FT%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Michael Jordan</td>
<td>1072</td>
<td>30.1</td>
<td>6.2</td>
<td>5.3</td>
<td>2.3</td>
<td>0.8</td>
<td>49.7</td>
<td>83.5</td>
</tr>
<tr>
<td>2</td>
<td>Magic Johnson</td>
<td>906</td>
<td>19.5</td>
<td>7.2</td>
<td>11.2</td>
<td>1.9</td>
<td>0.4</td>
<td>52.0</td>
<td>84.8</td>
</tr>
<tr>
<td>3</td>
<td>Oscar Robertson</td>
<td>1040</td>
<td>25.7</td>
<td>7.5</td>
<td>9.5</td>
<td>0</td>
<td>0</td>
<td>48.5</td>
<td>83.8</td>
</tr>
<tr>
<td>4</td>
<td>Kareem Abdul-jabbar</td>
<td>1560</td>
<td>24.6</td>
<td>11.2</td>
<td>3.6</td>
<td>0.7</td>
<td>2.0</td>
<td>55.9</td>
<td>72.1</td>
</tr>
</tbody>
</table>

We were surprised that Magic Johnson, who has won 5 NBA championships and 3 NBA Most Valuable Player (MVP) in his career, was not in the result. So we issued a why-not question \((Q_1, \{\text{Magic Johnson}\})\).

Using our algorithm, we got a refined query \(Q'_1\) in 10.2 ms (modifications are in bold face):

\[
Q'_1: \quad \text{SELECT P.Name}
\]
\[
\text{FROM Player P, Career C}
\]
\[
\text{WHERE P.PID = C.PID AND C.GP} \geq 1,000
\]
\[
\text{ORDER BY (} \frac{7}{5} \text{* C.PPG + } \frac{7}{8} \text{* C.RPG + } \frac{1}{2} \text{* C.APG}
\]
\[
+ \frac{1}{2} \text{* C.SPG} + \frac{1}{2} \text{* C.BPG} + \frac{1}{2} \text{* C.FG percent}
\]
\[
+ \frac{1}{2} \text{* C.FT percent) DESC}
\]
\[
\text{LIMIT 3}
\]

Its new top-k result was:

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>GP</th>
<th>PPG</th>
<th>RPG</th>
<th>APG</th>
<th>SPG</th>
<th>BPG</th>
<th>FG%</th>
<th>FT%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Michael Jordan</td>
<td>1072</td>
<td>30.1</td>
<td>6.2</td>
<td>5.3</td>
<td>2.3</td>
<td>0.8</td>
<td>49.7</td>
<td>83.5</td>
</tr>
<tr>
<td>2</td>
<td>Magic Johnson</td>
<td>906</td>
<td>19.5</td>
<td>7.2</td>
<td>11.2</td>
<td>1.9</td>
<td>0.4</td>
<td>52.0</td>
<td>84.8</td>
</tr>
<tr>
<td>3</td>
<td>Oscar Robertson</td>
<td>1040</td>
<td>25.7</td>
<td>7.5</td>
<td>9.5</td>
<td>0</td>
<td>0</td>
<td>48.5</td>
<td>83.8</td>
</tr>
<tr>
<td>4</td>
<td>Kareem Abdul-jabbar</td>
<td>1560</td>
<td>24.6</td>
<td>11.2</td>
<td>3.6</td>
<td>0.7</td>
<td>2.0</td>
<td>55.9</td>
<td>72.1</td>
</tr>
</tbody>
</table>

The refined query essentially hinted us that our original selection predicate C.GP \(\geq 1,000\) (the number of games played) eliminated Magic Johnson (who got brilliant records without playing a lot of games).

In this case, baseline algorithm Top-k could not answer the why-not question because it could not modify the predicate C.GP \(\geq 1,000\). Baselines Top-k-adapted and Staged returned the same refined query \((Q'_1)\) as our approach. However, they took much longer time (i.e., 258 and 96 ms, respectively).

Case 2 (Finding the top-3 players who performed best in year 2004). In this case, we first look up the top-3 players in year 2004:

\[
Q_2: \quad \text{SELECT P.Name}
\]
\[
\text{FROM Player P, Regular R, Playoffs O}
\]
\[
\text{WHERE P.PID = R.PID AND P.PID = O.PID}
\]
\[
\text{AND R.Year = 2004 AND O.Year = 2004}
\]
\[
\text{ORDER BY (} \frac{7}{5} \text{* R.PPG + } \frac{7}{8} \text{* R.RPG + } \frac{1}{2} \text{* R.APG}
\]
\[
+ \frac{1}{2} \text{* R.SPG} + \frac{1}{2} \text{* R.BPG} + \frac{1}{2} \text{* R.FG percent}
\]
\[
+ \frac{1}{2} \text{* R.FT percent) DESC}
\]
\[
\text{LIMIT 3}
\]

The initial result was: \{Dirk Nowitzki, Allen Iverson, Steve Nash\}. We wondered why Kobe Bryant was missing in the answer, so we posed a why-not question \((Q_2, \{\text{Kobe Bryant}\})\).

Our algorithm returned the following refined query in 18.1 ms:

\[
Q'_2: \quad \text{SELECT P.Name}
\]
\[
\text{FROM Player P, Regular R Playoffs O}
\]
\[
\text{WHERE P.PID = R.PID AND P.PID = O.PID}
\]
\[
\text{AND R.Year = 2004 AND O.Year = 2004}
\]
\[
\text{ORDER BY (} \frac{7}{5} \text{* R.PPG + } \frac{7}{8} \text{* R.RPG + } \frac{1}{2} \text{* R.APG}
\]
\[
+ \frac{1}{2} \text{* R.SPG} + \frac{1}{2} \text{* R.BPG} + \frac{1}{2} \text{* R.FG percent}
\]
\[
+ \frac{1}{2} \text{* R.FT percent) DESC}
\]
\[
\text{LIMIT 4}
\]

The refined query essentially hinted us that Kobe Bryant did not play PlayOff in year 2004. We checked back the data, and we were confirmed with the fact that Kobe Bryant’s host team, LA Lakers, failed to enter the Playoffs in that year. The result of \(Q'_2\) was: \{Allen Iverson, Dirk Nowitzki, Steve Nash, Kobe Bryant\}. As an interpretation, in addition to eliminating the PlayOff records, the refined weighting of \(Q'_2\) indicates that, in order to include Kobe Bryant in the result, we should weigh the players’ scoring/assisting/free throwing abilities higher than the other abilities.

In this case, baseline Top-k again could not work because it could not modify the query schema. Baseline Top-k-adapted returned the same refined query \((Q'_2)\) as
our approach, but took much longer time (576 ms). Baseline Staged returned the following refined query with higher penalty (0.37) and lower user satisfaction (2.1) compared to our approach (0.13 and 3.6, respectively):

\[ Q'^3: \]
\[
\text{SELECT Name FROM Player P, Regular R, Playoffs O WHERE P.PID = R.PID AND P.PID = O.PID AND R.Year = 2004 AND O.Year = 2004 ORDER BY (} \frac{1}{7} \times \text{R.PP} + \frac{1}{7} \times \text{R.RP} + \frac{1}{7} \times \text{R.FT percent) DESC LIMIT 7.}
\]

**Summary of Results:** Due to space limit, we do not include the details of case 3 and case 4 here. Briefly, case 3 is to find the top-3 players who performed best in their early career and case 4 is to find the top-3 players who have played more than 15 seasons of Playoffs, with the field goal percentage higher than 50 percent in each season. Readers can access the link of the online case study for details. In summary, baseline Top-k is not applicable to answer many why-not questions \((Q_1, Q_2, \text{ and } Q_3)\) because it cannot modify the SQL constructs. Baseline Top-k-adapted is able to return answers with penalty as good as our approach but it is slower by an order of magnitude. That is because it has to construct a restricted weighting space for each selection predicate. Baseline Staged runs slower and gives answers with poorer penalty and user satisfaction. That is because Staged is not solving the problem holistically but in a staged manner.

### 5.2 Performance Study

We use TPC-H to perform a performance study. We selected 10 TPC-H queries that can let us extend as top-k SPJA queries with minor modifications. The list of modifications of the selected queries is listed in Table 6.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Q₁</th>
<th>Q₂</th>
<th>Q₃</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Our Approach</strong></td>
<td>10.2 ms 0.39 3.5</td>
<td>18.1 ms 0.13 3.6</td>
<td>48.3 ms 0.09 3.7</td>
</tr>
<tr>
<td><strong>Top-k</strong></td>
<td>52.2 ms 0.25 2.4</td>
<td>—</td>
<td>29.5 ms 0.19 3.6</td>
</tr>
<tr>
<td><strong>Top-k-adapted</strong></td>
<td>258 ms 0.39 3.5</td>
<td>576 ms 0.13 3.6</td>
<td>566 ms 0.09 3.7</td>
</tr>
<tr>
<td><strong>Staged</strong></td>
<td>96 ms 0.39 3.5</td>
<td>158 ms 0.37 2.1</td>
<td>105 ms 0.09 3.7</td>
</tr>
</tbody>
</table>

Table 4 shows the parameters we varied in the experiments. The default values are in bold faces. The default weighting is \(\tilde{w}_0 = \left(\frac{1}{d}, \ldots, \frac{1}{d}\right)\), where \(d\) is the number of attributes in the ranking function. By default, the why-not question asks for a missing tuple/group that is ranked \((10 \times k_o + 1)\)-th under \(\tilde{w}_0\). The performance study includes two parts. Section 5.2.1 studies the various aspects of our proposed algorithm. Section 5.2.2 evaluates our algorithm with the baselines.

#### 5.2.1 Microbenchmark

**Effectiveness of Optimization Techniques.** In this experiment, we investigate the effectiveness of (i) excluding unnecessary selection condition (Section 3.3.1), (ii) the early stopping (Section 3.3.2) and (iii) skipping (Section 3.3.3) used in our algorithm. Fig. 7 shows the performance of our algorithm using only (i), only (ii), only (iii), all, and none, under the default setting. The effectiveness of the techniques is very promising when they are applicable (in particular, Lemma 2 is not used when the queries contain GROUP-BY). Without using any optimization technique, the algorithm requires a running time of roughly 1,000 seconds on these TPC-H queries. However, our algorithm runs about two to three orders faster when our optimization techniques are all enabled.

**Varying Data Size.** Fig. 8a shows the running time of our algorithm under different data size (i.e., scale factor of TPC-H). We can see our algorithm for answering why-not questions scales linearly with the data size for all queries.

**Varying \(k_o\).** Fig. 8b shows the running time of our algorithm using top-k SQL queries with different \(k_o\) values. In this experiment, when a top-5 query \((k_o = 5)\) is used, the corresponding why-not question is to ask why the tuple in rank 51st is missing. Similarly, when a top-50 query \((k_o = 50)\) is used, the corresponding why-not question is to ask why the tuple in rank 501st is missing. Naturally, when \(k_o\) increases, the time to answer a why-not question should also increase because the execution time of a progressive top-k SQL operation also increases with \(k_o\). Fig. 8b shows that our algorithm scales well with \(k_o\).

---

**TABLE 3**

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Q₁</th>
<th>Q₂</th>
<th>Q₃</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Our Approach</strong></td>
<td>10.2 ms 0.39 3.5</td>
<td>18.1 ms 0.13 3.6</td>
<td>48.3 ms 0.09 3.7</td>
</tr>
<tr>
<td><strong>Top-k</strong></td>
<td>52.2 ms 0.25 2.4</td>
<td>—</td>
<td>29.5 ms 0.19 3.6</td>
</tr>
<tr>
<td><strong>Top-k-adapted</strong></td>
<td>258 ms 0.39 3.5</td>
<td>576 ms 0.13 3.6</td>
<td>566 ms 0.09 3.7</td>
</tr>
<tr>
<td><strong>Staged</strong></td>
<td>96 ms 0.39 3.5</td>
<td>158 ms 0.37 2.1</td>
<td>105 ms 0.09 3.7</td>
</tr>
</tbody>
</table>
We next study the performance of our algorithm by posing why-not questions with missing tuples from different rankings (i.e., $r_o$). In this experiment, we set $k_o = 10$ and asked three individual why-not questions about why the tuple that ranked 101st, 501st, and 1001st, respectively, is missing in the result. Fig. 8c shows that our algorithm scales well with the ranking of the missing tuple. Of course, when the missing tuple has a worse ranking under the original weighting $\tilde{w}_o$, the progressive top-k SQL operation should take a longer time to discover it in the result and thus the overall running time must increase.

Varying Preference Option. We next study the performance of algorithm under different user preference on changing SPJ constructs, $k$ and $\tilde{w}$. These values can be system parameters or user specified as stated in Section 3.1. In Fig. 8d, it is good to show that our algorithm is insensitive to various preference options. In all cases, our algorithm can return answer very efficiently.

Varying the Number of Missing Tuples |$Y$|. We also study the performance of our algorithm by posing why-not questions with different numbers of missing tuples. In this experiment, a total of five why-not questions are asked for each TPC-H query. In the first question, one missing tuple that ranked 101th under $\tilde{w}_o$ is included in $Y$. In the second question, two missing tuples that respectively ranked 101th and 102th under $\tilde{w}_o$ are included in $Y$. The third to the fifth questions are constructed similarly. Fig. 8e shows that our algorithm scales linearly with respect to different size of $Y$.

Varying Query Size. In this experiment, we study the performance based on varying the number of predicates in each query. For example, when the number of predicates is controlled to 1, we randomly keep one selection predicate in the query and discard the rest. Fig. 8f shows that the performance is insensitive to the query size. That is because the search space of selection conditions $S_{QS^0}$ only depends on the number of attributes in the query schema $QS^0$ instead of the query size.

Varying T percent. We would also like to know how the performance and solution quality of our algorithm vary when we look for refined queries with different quality guarantees. Figs. 9a and 9b show the running time of our algorithm and the penalty of the returned refined queries when we changed from accepting refined queries that are within the best 10 percent ($|S| = 12$) to accepting refined queries that are within the best 0.1 percent ($|S| = 1.204$). From Fig. 9a, we can see that the running time of our algorithm increases when the guarantee is more stringent. However, from Fig. 9b, we can see that the solution quality of the algorithm improves when $T$ increases, until $T$ reaches 1 percent where further increases the sample size cannot yield any significant improvement.

Varying $Pr$. The experimental results of varying $Pr$, the probability of getting the best-$T$ percent of refined queries, are similar to the results of varying $T$ percent above. That is
because both parameters are designed for controlling the quality of the approximate solutions. In Figs. 9c and 9d, we can see that when we vary \( P \) from 0.1 to 0.9 (\( |S| = 460 \)), the running times increase mildly. However, the solution quality also increases gradually.

### 5.2.2 Comparison with Baselines

Fig. 10 shows the performance and penalty of our algorithm and the three baselines on TPC-H. Similar to our findings in case study, we observe that our approach is the most efficient. Top-k is comparable with our approach in terms of performance, but the quality of its answers is way poorer. Top-k-adapted is comparable with our approach in terms of answer quality, but its performance is the worst. Staged is poor in terms of both performance and answer quality. We also show a time breakdown of our algorithm. We see that Phase-2 dominates the execution time since it requires processing progressive top-k SQL operations for each candidate query.

Table 5 shows the maximum memory usage of our algorithm and the baselines. We observe that all algorithms consume roughly similar amount of memory. The memory is mainly used for holding the the join result \( J \) in memory, which is used in Phase-1 of all algorithms (Section 3.3), in order to avoid repeated computation of the same join result based on \( QS' \) in the subsequent phase.

Lasty, we indeed also implemented another baseline algorithm, Exact, which leverages a quadratic programming (QP) solver to solve the why-not problem mathematically. Unfortunately, this exact solution ran too slow and thus we only ran experiments for TPC-H Q2 on 1Gb data. In that experiment, the best refined query returned by Exact incurs a penalty of 0.298. However, Exact took 30 days to find that. In contrast, our algorithm is able to find a refined query with 0.31 penalty, but in 1.45 seconds.

### 6 Conclusion

In this paper, we have studied the problem of answering why-not questions on top-k SQL queries. Our target is to give an explanation to a user who is wondering why her expected answers are missing in the query result. We return to the user a refined query that can include the missing expected answers back to the result. Our case studies and experimental results show that our solutions efficiently return very high quality solutions. In future work, we will study this issue on queries involving non-numeric attributes.

### Acknowledgments

The research was supported by grants 521012E, 520413E, 15200715 from Hong Kong RGC. Zhian He is the corresponding author.
REFERENCES


