QUERY BY OUTPUT

TRAN QUOC TRUNG

B.Eng. in Computer Science

HCM City University of Technology

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgements

This thesis would not have been finished without the support and encouragement of many people. I would like to reserve this section to express my gratitude to all of them.

The course of completing this thesis was a wonderful opportunity of interaction with my supervisor, Prof. Chee-Yong Chan. His guidance, support, and dedication to research have been a great inspiration to me. He had spent countless hours not only of working days but also weekends and public holidays to help me develop ideas, elaborate on details, write and present technical works. His insights and suggestions were invaluable for my work. Whenever I encountered difficulties in research or even in personal life, I always felt comfortable to approach him for advice. I am deeply grateful to him.

My deep gratitude also goes to Prof. Srinivasan Parthasarathy, whom I had a chance to collaborate with on my first research paper. He had many insightful comments on my work. I would also like to thank Prof. Kian-Lee Tan and Prof. Wynne Hsu for being in my thesis evaluation committee and providing constructive feedback to refine my work. I would also say thank to Prof. Hon Wai Leong for helping me in the first year at NUS.

I wish to thank the following people: Hoang Viet Tung, Do Huy Hoang, Cao Thanh Tung, Htoo Htet Aung, Nguyen Thi Hong Duyen, all the members of Database Research Laboratories, many friends in NUS, and many Vietnamese friends for the open discussions, valuable assistance, and enjoyable hours we spent together at the leisure time.

Last but not least, I wish to thank my family for taking care and supporting me over the time. It is to my mother whom I dedicate this thesis.

Table of Contents

Summary

While database system research has made tremendous advances on functionality and performance related issues over the years, research on improving database usability has not attracted as much attention as it deserves. In this work, we propose a novel data-driven approach, called *Query by Output (*QBO*)*, targeted at improving the usability of database management systems. The central goal of *Query by Output* is as follows: given a database D and a result table $T = Q(D)$, which is the output of some query Q on D, the goal of QBO is to construct an alternative query Q' such that the output of query Q' on database D is equal to $Q(D)$. We consider the following three variants of QBO.

In the first variant of QBO, the input query Q may be known or unknown, and the result table T contains a set of specific tuples. One useful application of this variant is to help users better understand their query results by augmenting the result of a query Q (w.r.t. a database D) with *instance-equivalent queries* Q', each of which produces the same result as Q (w.r.t. D) and thus describing alternative characterizations of tuples in $Q(D)$.

The second variant of QBO requires the input query Q to be explicitly given and the result table to be in the form $T = Q(D) \cup S$, where S is a non-empty set of expected tuples that are missing from $Q(D)$. The problem is to derive some *refined query* Q' of Q such that $Q'(D)$ includes all the tuples in $Q(D)$ as well as the missing tuples in S. This variant of QBO presents a *new paradigm* for explaining why a set of tuples is missing in the result of a query Q w.r.t. a database D by automatically generating one or more refined query Q', whose result includes both the original query

result and the missing tuples.

The third variant of QBO takes the following as inputs: (1) a database D, (2) a query Q that is partially specified (e.g., only the from-clause and the join predicates of Q are specified), and (3) a set C of aggregation constraints that must be satisfied by the query result of each derived query Q' from Q. The goal is to either (1) *evaluate* Q by returning a set of tuples $S_{ans} \subseteq Q(D)$ satisfying all the constraints in C, or (2) *instantiate* Q into one or more complete relational queries Q' such that the execution of Q' on D satisfies all the constraints in C. One useful application of this setting is in the *Targeted Query Generation* problem to generate test queries for database testing.

List of Tables

List of Figures

Chapter 1

Introduction

While database system research has made tremendous advances on functionality and performance related issues over the years, research on improving *database usability* has not attracted as much attention as it deserves [23]. Since databases are currently hard to design, modify, and query, there are many expected functionalities that database systems should provide for users to store and retrieve information easily and efficiently. For instance, database systems should provide an *explain capability* for users to seek clarifications on unexpected query results. The necessity for this functionality comes from the fact that responses to queries might not contain information that users want, or contain the results that are unexpected by users [15]. In another example, database systems should provide novice users with a simple method to query the systems (e.g., using keyword search with flexible semantics), while still providing expert users with tools that maximize their productivity (e.g., tools for database testers to automatically generate test queries satisfying certain kinds of properties). Some recent approaches have been proposed towards improving database usability. One example is the idea of Précis queries [28, 44], which aims to augment a user's query result with other related information (e.g., relevant tuples from other relations). Another example includes some explanation models (e.g., [9, 20, 22, 33]) to help users seek clarifications on the absence of expected tuples in the query results.

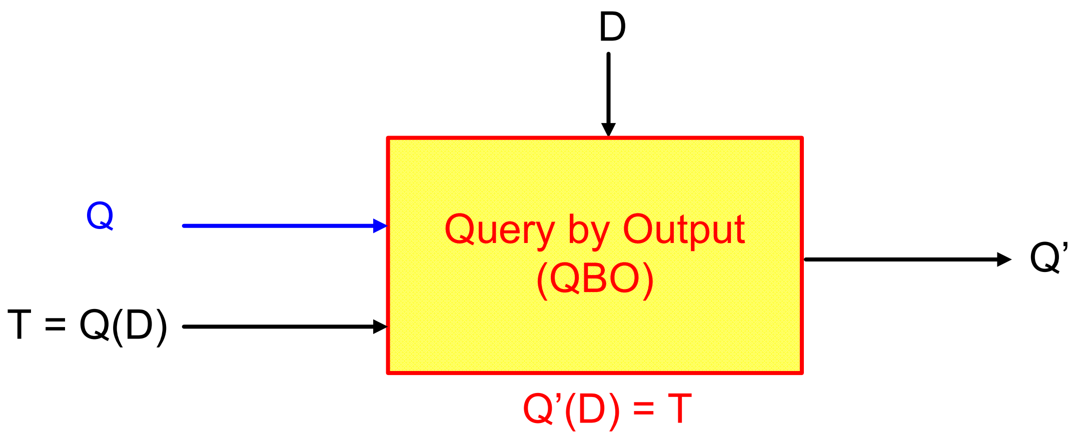


Figure 1-1: Problem Statement of Query by Output

1.1 An Overview of Query by Output

In this work, we propose a novel data-driven approach, called *Query by Output* (QBO), targeted at improving the usability of database management systems. The central goal of *Query by Output* is as follows: given a database D and a result table $T = Q(D)$, which is the output of some query Q on D, the goal of QBO is to construct an alternative query Q' such that the output of query Q' on database D is equal to $Q(D)$. We refer to Q as the *input query*, D as the *input database*, $T = Q(D)$ as the *given result table*, and Q' as the *output query*. The problem statement of QBO is visualized in Figure 1-1.

We investigate three variants of QBO corresponding to different settings of the two over three input parameters, including the input query Q and the given result table T. The main contributions of our work are summarized in Table 1.1.

QBO Problem	**Parameters**	
	Input query Q	**Given result table T**
The first variant (published in [49])	- Q is known - Q is unknown	$T = Q(D)$ T is a set of specific tuples
The second variant (published in [48])	Q is known	$T = Q(D) \bigcup S$ S is a set of tuples that are not present in $Q(D)$
The third variant (published in [50])	Q is partially specified	T is a set of constraints that must be satisfied by the query result of each derived query Q'

Table 1.1: Summary of Our Work's Contributions

In the first variant of QBO, the input query Q may be known or unknown, and the result table T contains a set of specific tuples. One useful application of this variant is to help users better understand their query results by augmenting the result of a query Q (w.r.t. a database D) with alternative queries Q', each of which produces the same result as Q (w.r.t. D) and thus describing alternative characterizations of tuples in $Q(D)$ (Section 1.2).

The second variant of QBO requires the input query Q to be explicitly given and the result table to be in the form $T = Q(D) \cup S$, where S is a non-empty set of expected tuples that are missing from $Q(D)$. The problem is to derive some *refined query* Q' of Q such that $Q'(D)$ includes all the tuples in $Q(D)$ as well as the missing tuples in S[1]. This variant of QBO presents a *new paradigm* for explaining why a set of tuples is missing in the result of a query Q w.r.t. a database D by automatically generating one or more refined query Q', whose result includes both the original query result and the missing tuples (Section 1.3).

The third variant of QBO takes the following as inputs: (1) a database D, (2) a query Q that is partially specified (e.g., only the from-clause and the join predicates of Q are specified), and (3) a set C of aggregation constraints that must be satisfied by the query result of each derived query Q' from Q. The goal is to either (1) *evaluate* Q by returning a set of tuples $S_{ans} \subseteq Q(D)$ satisfying all the constraints in C, or (2) *instantiate* Q into one or more complete relational queries Q' such that the execution of Q' on D satisfies all the constraints in C. One useful application of this setting is in the *Targeted Query Generation* problem to generate test queries for database testing (Section 1.4).

Essentially, the second variant of QBO can be formulated as an instance of the first variant of QBO as follows. By treating the query's result $Q(D)$ together with the missing tuples S as the output result of some unknown query, the first variant of QBO will derive alternative queries Q' such that $Q'(D) = Q(D) \cup S$. The alternative query Q' in this case can be considered as the refined query to explain why the set of tuples S is miss-

[1]For certain cases where S involves a constraint specification, the attribute values associated with the constraints could be different between $Q'(D)$ and $Q(D)$.

ing from $Q(D)$. Correspondingly, the first variant of QBO can also be formulated as an instance of the second variant of QBO as follows. By considering all tuples in the given result table $T = Q(D)$ as the missing tuples (i.e., $S = Q(D)$) and constructing the input query to return empty result, the second variant of QBO will generate refined queries Q' such that $Q'(D) = Q(D)$. Indeed, our proposed solutions for these two settings of QBO can be applied in both cases. We will discuss which approaches should be used for each of these two settings of QBO in Section 5.5.4.

1.2 Deriving Instance Equivalent Queries

The first variant of *Query by Output* (QBO) is a novel data-driven approach that aims to derive interesting query-based characterizations of a given input query Q w.r.t. a database D and tuples in its result. This setting is designed to work in the "reverse direction" with the conventional querying that takes an input query Q and computes its output, denoted by $Q(D)$, w.r.t. an input database D. In contrast, the basic idea of QBO is to take as input the query result $Q(D)$ of some query Q, and compute a set of queries $Q'_1, \cdots, Q'_n$ such that each $Q'_i(D)$ is (approximately) equal to $Q(D)$. We say that two queries Q and Q' are *instance-equivalent* w.r.t. a database D (denoted by $Q \equiv_D Q'$) if $Q(D)$ and $Q'(D)$ are equal. In this setting, the input query Q may be known or unknown.

There are several scenarios where this setting of QBO is useful. In the following discussions, we highlight some of the use-case scenarios of this setting.

Database Usability. The most obvious application of QBO is in database usability. Consider the scenarios when a user wants to evaluate a query Q on a database D. Instead of simply returning the query result $Q(D)$ to the users, the database system can also apply QBO to derive instance-equivalent queries Q' of Q that describe alternative characterizations of tuples in the query result of Q.

Example 1.1 *Consider the relation Movie($\underline{title}$, year, gross-revenue, director) in a movie* *database. Suppose a user submitted a query Q_1 to find movies that are directed by "James Cameron" since* 1997*. The query result $Q_1(D)$ includes two movies: "Avatar" and "Titanic".* QBO *can be applied in this case to provide alternative characterizations of the two movies in $Q_1(D)$. The inputs to* QBO *include the movie database D, the query Q_1, and the given result table $Q_1(D) = \{$("Avatar"), ("Titanic")$\}$. There is an insightful query that is instance-equivalent to Q_1 w.r.t. D, and specifies "select the movies that are the top-*2 *high gross-revenue movies".* □

As the above example illustrates, the ability to return instance-equivalent queries for a given query Q can reveal interesting properties of the query result $Q(D)$. In addition, unusual or surprising IEQs can be useful for uncovering hidden relationships among the data. In several instances, simpler or easier to understand relationships may be uncovered; this can again aid in the understanding of the data contained within the complex database. As an example, consider a skyline query Q_2 looking for people with maximal capital gain and minimal age in Adult data set[2]. An instance-equivalent query Q'_2 of Q_2 provides a simplification of this query: the people selected by this skyline query are very young ($age \leq 17$) and have low capital gain (< 5000), or they have very high capital gain (> 27828) and work in the protective service.

Besides providing alternative characterizations of the query results, IEQs can also help users to better understand the database schema. Since many enterprise data schema are very complex and large, the part of the database schema that is referenced by the user's query may be quite different from that referenced by an IEQ. The discovery of this alternative "part" in the schema to generate an instance-equivalent query can aid the user's understanding of the database schema, or potentially help refine the user's initial query.

Example 1.2 *Consider the baseball data set[3] and a query Q_3 that finds managers of "CIN" team during the years from* 1982 *to* 1988*. This query involves the join between*

[2]http://archive.ics.uci.edu/ml/datasets/Adult
[3]http://baseball1.com/statistics/

two relations Manager and Team. An instance-equivalent query Q'_3 of Q_3 reveals that some of these managers were also the players of "CIN" team at the same time they managed the team. The IEQ Q'_3 has revealed the alternative schema part that involves the joins from a different set of relations (Manager, Team, Master, and Batting), and provided users with useful information about these player-managers. □

QBO can also be applied to provide succinct query-based *explanation* for each partition that is produced by some data partitioning algorithms. As a specific example, consider a relational-cloud system, *Schism*, proposed in [13] that utilizes database partitioning to scale a single database to multiple nodes. *Schism* partitions data to place different partitions into different nodes in such a way that most transactions should completely touch data at one node. *Schism* needs a succinct representation of these partitions to route SQL queries into the correct place. The idea of QBO to derive predicate-based characterization of each partition can be applied in this system [13].

Database Security. QBO may also have interesting applications in database security, where attackers who have some prior domain knowledge of the data may attempt to derive sensitive information. For example, if an attacker is aware of the existing correlation structure in the data, they can easily use this information to formulate two or more separate queries that on paper look very different (e.g., using different selection criteria), but in reality may be targeting the same set of tuples in the database. Such sets or groups of queries can potentially be used to reverse-engineer the privacy preserving protocol in use. Subsequently, sensitive information can be gleaned. As a specific example, consider a protocol such as ϵ-diversity [54], which relies on detecting how similar the current query is with a previous set of queries (history) answered by the database, to determine if the current query can be answered without violating the privacy constraints. The notion of similarity used by such methods relies primarily on the selection attributes, and thus such protocols will fail to recognize IEQs that use different selection attributes. Privacy in such protocols will then be breached. Automatically recognizing such IEQs via the methods proposed in this work and subsequently leveraging this information to enhance such protocols may provide more stringent protection against such kinds of attacks.

Data Exploration & Analysis. Another important class of QBO applications is in scenarios where the input query Q is unknown. Specifically, consider a view V (defined on a database D) which may have been derived manually (e.g., a user selected some tuples of the database of interest to her), or by an application program that is no longer available (e.g., the program is no longer maintained or is lost). Therefore, given only the view result V on the database D, it will be very useful to be able to derive instance-equivalent queries Q' of the unknown query for V (i.e., $V = Q'(D)$) that describe the characteristics of the tuples in V. In data exploration, such scenarios are more common where the documentation and meta-data for the data sets being analyzed are incomplete, inaccurate, or missing. As an example, in AT&T's Bellman project [24], the data set made available to data analysts is often in the form of a delimited ASCII text file representing the output result of some query, where the query is not available for various reasons. Clearly, it will be useful to reverse engineer the queries to help make sense of the data before performing further analysis tasks (e.g., data mining).

Materialized View Maintenance. Another useful application of QBO is in materialized view maintenance, where a view V (defined on a database D) may have been derived manually or by an application program that is no longer available. When D is modified into D', the challenge is how to propagate the data updates to the view V. One solution here is to reverse-engineer a view definition Q that captures the relationship between V and D, and then apply Q on D' to derive the modified view for V [43].

Data Integration. In data integration systems, the goal is to combine data residing at different sources to provide users with a unified view of these data [30]. The *global-as-view* integration approach requires that the global schema be expressed in terms of the data sources, which necessitates a query over the global schema to be reformulated in terms of a set of queries over the data sources. Thus, the QBO problem in this context is given a result table that is generated by the integration system to find the instance-equivalent query that involves the union of sub-queries over the data sources.

1.3 Explaining Why-Not Questions

Our first variant of QBO to provide additional useful information about tuples in the query result aims to address part of the problem when responses to queries do not contain information that users want, or contain the results that are unexpected by users. In this section, we present the second setting of QBO to address users' concerns about unexpected query results in the form of explaining why-not questions.

The second variant of QBO targets at providing an *explain capability* for users to seek clarifications on query results, a useful feature that is missing from today's database systems. Although most database systems today provide an explain functionality to help database administrators understand and tune the performance of unexpected slow-running queries (e.g., SQL *EXPLAIN* command), there is no similar higher-level explain feature available to help end users understand the unexpected results in their query outputs. There are two types of unexpected query results that are of interest: (1) the presence of unexpected tuples, and (2) the absence of expected tuples (i.e., missing tuples). Clearly, it would be very helpful if users could pose follow-up *why* questions (i.e., why is a certain tuple in the result) or *why-not* questions (i.e., why is a certain tuple missing from the result) to seek clarifications on unexpected query results. While the why questions can be addressed by applying established data provenance techniques [47], the problem of explaining the why-not questions has received very little attention [9].

Consider the following SQL query to find the recent high-scoring NBA players from the NBA statistics[4]: SELECT P.name FROM Player P, Regular R WHERE P.pID = R.pID AND R.year > 2000 AND R.pts > 2400. Among the players returned by the query are many expected well-known NBA superstars such as "LeBron James" and "Kobe Bryant". However, the user is surprised to find that the superstar "Rick Barry" is

[4]http://www.basketballreference.com/.

absent from the query result. At this point, the user could try to figure out for himself an explanation for the missing tuple by relaxing at least one selection predicate (e.g., adjusting the year to 1990 or lowering the points to 2000) to see if Barry satisfies the revised query. Clearly, such a manual trial-and-error approach of seeking explanation is rather tedious involving possibly many rounds of query refinement. Moreover, the user could end up over-relaxing his refined query and obtaining many more additional result tuples than just the tuple for Barry. Thus, it would be very helpful to the user if she could simply pose a single why-not question to the database system to seek an explanation for why "Rick Barry" is not in the result.

There are two main models for explaining why-not questions. One natural explanation model for missing tuples is to identify the query operator(s) that is responsible for eliminating the missing tuples from the result [9]. Thus, for the above example, a possible explanation is to identify the selection operator on the *year* attribute as the "culprit" operator. For applications where a query result is computed by a workflow of black-box processing steps, the ability to pinpoint the step that is responsible for the missing tuples could be the most informative available explanation.

However, in general, an even more helpful explanation can go beyond merely identifying the culprit step/operator, and actually suggests one or more ways to "fix" the original query such that the missing tuples become present in the result. Continuing with the example, a more informative explanation would be a refined query that changes the selection predicate on *year* to "R.year > 1970". In this way, not only does the system reveal the culprit operator to the user, it also explicitly shows the user how to revise the original query to obtain the expected tuples. The automatic generation of refined queries to explain unexpected query results can be useful even for applications that interact with users via a form-based web-interface, where the SQL queries being issued to the database are generated by a middleware component based on the completed forms. Basically, what is needed is a component to map a refined query back to an interface-based explanation. For example, an interface-based explanation could inform the user that had she clicked on button X on the form and selected item Y from the pulled-down menu, the expected missing tuple would have been included in the result.

A second model that has been proposed explains a missing tuple t in terms of modifications to the database such that t appears in the query result w.r.t. the modified database [20, 21, 22]. This model was proposed in the context where some of the data in the database are extracted from untrusted information sources that may not be accurate. Thus, the intuition of this model is to explain in terms of how to modify some of the untrusted data in order to produce the missing tuple. Clearly, this explanation model is very flexible if arbitrary modifications to the database are allowed to derive the missing tuples. However, this model may not be applicable in applications where all the data stored are trusted (e.g., enterprise databases), and where it may not be meaningful to make arbitrary changes to the stored data.

In this work, we propose a new explanation model that is based on automatically generating one or more *refined query*, whose result includes both the original query's result as well as the missing tuples. With respect to the framework of QBO, the input query to this setting is the original query, and the given result table is the union of the result of the input query and the set of missing tuples (as shown in the second row in Table 1.1). Our proposed model goes beyond identifying culprit query operator(s) (in contrast to the first explanation model), and actually recommends refined queries, instead of data changes (unlike the second explanation model), to "fix" missing tuples. It is desirable for a refined query to be as similar as possible to the original input query by making only minimal relaxations to appropriate selection predicates in the query. However, doing so might not always generate the desired missing tuples, as the user's original query might actually be focused on the "wrong" part of the database schema and needs to be reformulated. Thus, our proposed explanation strategy will try to generate minimally modified refined queries to account for the missing tuples whenever possible, and resort to more drastic query reformulation if minimally refined queries do not exist. Besides handling why-not questions for select-project-join (SPJ) queries, our approach can also explain why-not questions for SPJ queries with aggregation (SPJA queries) that are not addressed by any of the previous explanation models. The later pieces of work in [20, 33] also handle why-not questions for SPJA queries.

The following three examples illustrate the capabilities of our proposed approach. The first example illustrates the need to sometimes refine query beyond simply relaxing selection predicates by reformulating the query to retrieve from different relations in the database schema.

Example 1.3 *Consider a flight database, which includes two relations budget_airline and regular_ airline that describe airfare information for budget and regular airlines, respectively. Suppose a user wants to buy a cheap airticket for vacation travel in July with the criteria that the departure city is in Singapore and the ticket price is at most* 400*. The user issues a query on the budget_airline relation to find all the destination cities that meet his requirement, and is surprised to learn that "Shanghai" is not listed in the result even though one of his colleagues has recently booked a cheap airticket to "Shanghai". It could well be that there is no available air tickets from Singapore to Shanghai with the budget airlines; however, there are promotion cheap airtickets available with the regular airlines that meet his requirement. Thus, in this case, simply relaxing the predicates in the original input query would not help to generate any explanation. Instead, the refined query needs to be reformulated on both the budget_airline and regular_airline relations.* □

The next example illustrates why-not queries with constraints on aggregated values.

Example 1.4 *Consider the following query to find the average points scored by high-scoring recent basketball players: SELECT P.name, AVG(R.pts) FROM Player P, Regular R WHERE P.pID = R.pID AND R.pts* $>$ 2000 *AND R.year* $\geq$ 1994 *AND R.year* $\leq$ 2000 *GROUP BY P.name. The result contains two tuples (Michael Jordan,* 2200*) and (Gary Payton,* 2800*). The user might be expecting Jordan's average score to be higher, and would like to seek an explanation for why Jordan's average score is not* *higher than* 3000*. Our approach can process such why-not questions involving selection constraints on an aggregated value. It turns out that Jordan actually did not perform so well during* 1994*, and a refined query that replaces "R.year* ≥ 1994*" with "R.year* ≥ 1995*" would explain the user's why-not question. Our approach can also support this kind of why-not questions for missing tuples. For example, the user could ask why "Wilt Chamberlain" is not in the result. Or even more specifically, the user asks why "Wilt Chamberlain" is not in the result with an average score of at least* 3000. □

The final example illustrates more complex why-not queries involving relative comparisons of aggregated values.

Example 1.5 *Suppose Professor P issues the following query to check the academic performance (in terms of average scores) of his students: SELECT G.name, AVG (G.score) FROM Grade G GROUP BY G.name. P is surprised to find the tuples (Alice,*70*) and (Bob,* 90*) in the result, as he has expected Alice to perform better than Bob. Thus, P would like to ask why Alice's average score is not higher than Bob's. Our approach can handle such sophisticated why-not questions that involve comparisons among multiple aggregated values in the results. A possible explanation for this why-not question is the following refined query: SELECT G.name, AVG (G.score) FROM Grade G WHERE G.dept = "CS" GROUP BY G.name, which explains that Alice indeed performs better than Bob if the average scores were computed for courses offered by the "CS" department.* □

1.4 Instantiation & Evaluation of Partial Queries

The third setting of QBO works towards providing novice users with a flexible method to query the systems, while still providing expert users with tools that maximize their productivity. We introduce the concept of *partial queries*, which is a class of extended relational queries that allows flexible data retrieval using aggregation constraints. In contrast to the conventional relational queries where there is a unique set of tuples satisfying each query, a partial query generally has multiple possible results arising from the application of the aggregation constraints.

Informally, a partial query $Q = (Q_{base}, C_{ans})$ consists of two components. The first component is a base query Q_{base}, which is a conventional relational query that returns a set S_{base} of tuples to serve as the base data for the partial query. The second component is a set C_{ans} of constraints that must be satisfied by each result of Q, which is a subset of S_{base}. With respect to the framework of QBO, Q_{base} corresponds to the input query and C_{ans} corresponds to the given result table.

Example 1.6 *Suppose that Alice wants to download a set of MP3 files into her iPhone from a music database containing a relation* $Song(title, genre, album, artist, filesize, length)$ *satisfying the following three requirements: (1) the songs must be rock music, (2) the total size of the files must be as large as possible but not exceeding* 500*MB, and (3) the total number of different artists for the downloaded songs should be between* 5 *and* 7 *(for diversity). Observe that Alice's retrieval request is not expressible in conventional relational query languages, but can be expressed using the following partial query Q. The base query* Q_{base} *of Q retrieves the set* S_{base} *of songs satisfying condition (1); i.e.,* Q_{base} *corresponds to the following SQL query: SELECT * FROM Song WHERE genre = "rock". The constraint set* C_{ans} *of Q contains conditions (2) and (3). Each subset of* S_{base} *that satisfies C is a possible answer to Q.* □

Besides the basic sum and count aggregation constraints illustrated by the example above, partial queries also support *content constraints* to indicate the presence of certain attribute values (or tuples) in each result. For example, Alice could indicate that each result must contain some song by "The Beatles" and some song by "Bob Dylan". Another useful type of constraints supported is *group-by constraints* on the query result. Continuing with the previous example, Alice could also specify a "group-by-count" constraint, requiring that there must not be more than two songs that belong to the same album; or a "group-by-sum" constraint, requiring that the total length of the songs from the same album cannot exceed 30 minutes.

Special cases of partial queries, which have restrictions on the type of constraints allowed, have been proposed for various applications including optimization problems in business applications [18], multiple-choice Knapsack problem (e.g., government budgeting with demands in different sectors) [26], and student course planning applications [2].

As the above example illustrates, a partial query could have multiple possible answers depending on the number of subsets of S_{base} that satisfies the constraints in C_{ans}. Given this property of partial queries, there are two different modes to process a partial query resulting in two different use cases for partial queries. The first processing mode is to *evaluate* the results of a partial query to return any answer, all answers, or top-k answers based on some ranking criteria. This mode is very useful for data retrieval applications.

The second processing mode is to *instantiate* a partial query into one or more complete relational queries, where the result of each instantiated query is an answer to the partial query. We say that a relational query Q_{inst} is an *instantiation* of a partial query $Q = (Q_{base}, C_{ans})$ w.r.t. a database D if (1) the result of Q_{inst} on D satisfies all the constraints in C_{ans}, and (2) Q_{inst} is derived from Q_{base} by modifying its selection predicates; the modifications include changing the constants in the existing selection predicates of Q_{base} and/or introducing additional selection predicates.

The instantiation processing mode has application in the Targeted Query Generation (TQG) problem to generate targeted queries for database testing [7, 35]. In the testing of database systems, it is important to be able to generate test queries that satisfy certain cardinality constraints on the intermediate subexpressions of the queries. The TQG problem was first studied by Bruno et al. [7], and subsequently generalized by Mishra et al. [35]. The inputs to the TQG problem include a query Q, a database D, and a set of cardinality constraints C on subexpressions of Q; and the objective is to derive a new query Q' from Q (by modifying the constant values in Q's selection predicates) such that the execution of Q' on D satisfies all the cardinality constraints in C.

Example 1.7 *As an example of a TQG problem specification, consider the following input query Q and set C of cardinality constraints. Q is the following SQL query:*

*SELECT * FROM R_1, R_2, R_3 WHERE p_1 AND p_2 AND p_3 AND $j_{1,2}$ AND $j_{2,3}$,*

*where each p_i is a conjunction of selection predicates on relation R_i, and each $j_{i,k}$ is a join predicate between R_i and R_k. C contains the following cardinality constraints on Q and two subexpressions, Q_1 and Q_2, of Q: (1) $|Q_1| = n_1$ where Q_1 = SELECT * FROM R_1 WHERE p_1, (2) $|Q_2| = n_2$ where Q_2 = SELECT * FROM R_1, R_2 WHERE p_1 AND p_2 AND $j_{1,2}$, and (3) $|Q| = n_3$.* $\square$

Note that in the TQG problem, the specification of a subexpression Q_i of the query Q requires only identifying the subset of relations in Q that appear in Q_i (i.e., its SQL query's FROM-clause). The selection and join predicates in Q_i are simply all the applicable predicates from Q involving only the relations in Q_i, and all the attributes are projected in Q_i. This property of subexpression specifications is due to the fact that the subexpressions are intended to denote sub-plans of Q's query plan [35].

By extending the definition of partial queries to support constraints on subexpressions of the base query, a TQG problem can be specified in terms of a partial query $Q = (Q_{base}, C_{ans})$, where Q_{base} is equal to the TQG's input query and C_{ans} is equal to the TQG's set of cardinality constraints. For the TQG problem, each constraint in C_{ans} is a cardinality constraint specifying the number of intermediate tuples produced by a query sub-plan. Thus, an answer to a TQG problem corresponds to an instantiation of a partial query.

We use `PQE` to refer to the problem of evaluating partial queries, and `PQI` to refer to the problem of instantiating partial queries. Between `PQE` and `PQI`, the problem of `PQI` follows the spirit of `QBO`'s framework and the problem of `PQE` is a by-product of manipulating partial queries. However, we expect `PQE` to be the most common usage of partial queries, since `PQE` is very useful for data retrieval applications.

1.5 Summary of Contributions

It has recently been asserted that the usability of a database is as important as its capability [23]. Providing tools for users to understand the database schema, the hidden relationships among attributes in the data, as well as for users to retrieve information easily and efficiently plays an important role in this context. Subscribing to these viewpoints, we make the following contributions in this work:

- Our first contribution is to introduce the novel problem of *Query by Output* (`QBO`) that can enhance the usability of database systems. We propose a solution (`TALOS`) that models the `QBO` problem as a data classification task with a unique property that we term *at-least-one semantics*, which is inherent in the derivation of the instance-equivalent queries (IEQs). To handle data classification with this new semantics, we develop a new dynamic class labeling technique. In addition to the basic framework, we design several optimization techniques to reduce processing overhead, and introduce a set of criteria to rank order output queries by various notions of utility. Our experimental evaluation of `TALOS` demonstrates its efficiency and effectiveness in generating interesting IEQs. We also generalize the first setting of `QBO` with the following three additional challenges: (1) the original query is not given as part of the input, (2) the derived queries are more expressive and go beyond the simple Select-Project-Join query fragment, and (3) there are multiple database versions. We present a generalized approach (`REQUERE`) to address these issues, and demonstrate its effectiveness and efficiency with an experimental evaluation on real data sets.

- Our second contribution is to propose a new paradigm for explaining a why-not question that is based on automatically generating a refined query, whose result includes both the original query's result as well as the user-specified missing tuples. In contrast to the existing explanation models, our approach goes beyond merely identifying the "culprit" query operator(s) responsible for the missing tuples, and is useful for applications where it is inappropriate to modify the database to obtain missing tuples. With this new paradigm for explaining why-not question, we introduce a new framework, named ConQueR, to explain why-not questions based on automatically generating refined queries. We propose novel algorithms to not only handle basic SPJ queries, but also more sophisticated SPJA queries that involve constraints or comparisons among aggregated values. We demonstrate the usefulness of our paradigm by comparing against the two existing explanation models on both synthetic and real data sets, and show the efficiency of ConQueR by a performance comparison against TALOS, the classification-based approach for the first variant of QBO to generate instance-equivalent queries.

- Our third contribution is to introduce the concept of partial queries and two useful modes of processing partial queries, including evaluating partial queries (PQE) and instantiating partial queries (PQI). With respect to PQE, we first prove that even for partial queries with only count constraints, the PQE problem is already NP-complete *in the strong sense*. Although there are several problem formulations that correspond to special cases of partial queries and are solvable with polynomial [2] or pseudo-polynomial complexity [18, 26], there remains two open questions. First, there is the question of whether there are other non-trivial special cases of partial queries that are amenable to polynomial/pseudo-polynomial evaluation algorithms. Second, the problem of evaluating general partial queries with arbitrary constraints (sum, count, optimization, group-by, content) has not been addressed to the best of our knowledge. In this work, we address these two questions by presenting two novel evaluation algorithms, DP and Greedy, respectively. The first algorithm, DP, is a pseudo-polynomial algorithm for evaluating partial queries with multiple sum constraints and at most one of either count, content, or group-by constraint. The second algorithm, Greedy, is a heuristic approach for evaluating general partial queries with any combination of constraints. Greedy tries to find a solution that satisfies all the specified constraints but may return an approximate solution that meets only some of the constraints.

With respect to PQI, we propose two approaches to instantiate partial queries. Our first approach is a data-driven approach, which is more general than the state-of-the-art approach in [35]. In contrast to [35], which produces the target test query by modifying only the constants in the input query's selection predicates, our approach of generating instantiated queries can also add new selection predicates to the instantiated queries. This flexibility is important, as it may not be always possible to generate an output query that satisfies all the cardinality constraints by merely modifying the existing selection predicates in the input query. Our second approach is a more efficient sampling-based method, but the generated instantiated query might not satisfy all the constraints.

Parts of the materials of this work on TALOS, ConQueR, and evaluating partial queries were previously published in [49], [48], and [50], respectively.

1.6 Thesis Organization

The remaining of this thesis is organized according to the techniques that we have introduced to solve the three problem settings of QBO. The associated chapters that discuss each of these problems are depicted in Figure 1-2. We summarize the main contents of these chapters in more details next:

- *Related Work*: Chapter 2 presents the related work of each of the three settings of QBO.
- *Deriving Instance-Equivalent Queries*: Chapter 3 describes our classification-based approach, TALOS, to solve the first variant of QBO to derive instance-equivalent queries (IEQs) of a given input query w.r.t. a given database, where the derived IEQs are in the Select-Project-Join relational fragment. Chapter 4 then presents a generalized system of TALOS, named REQUERE, which enhances TALOS along the

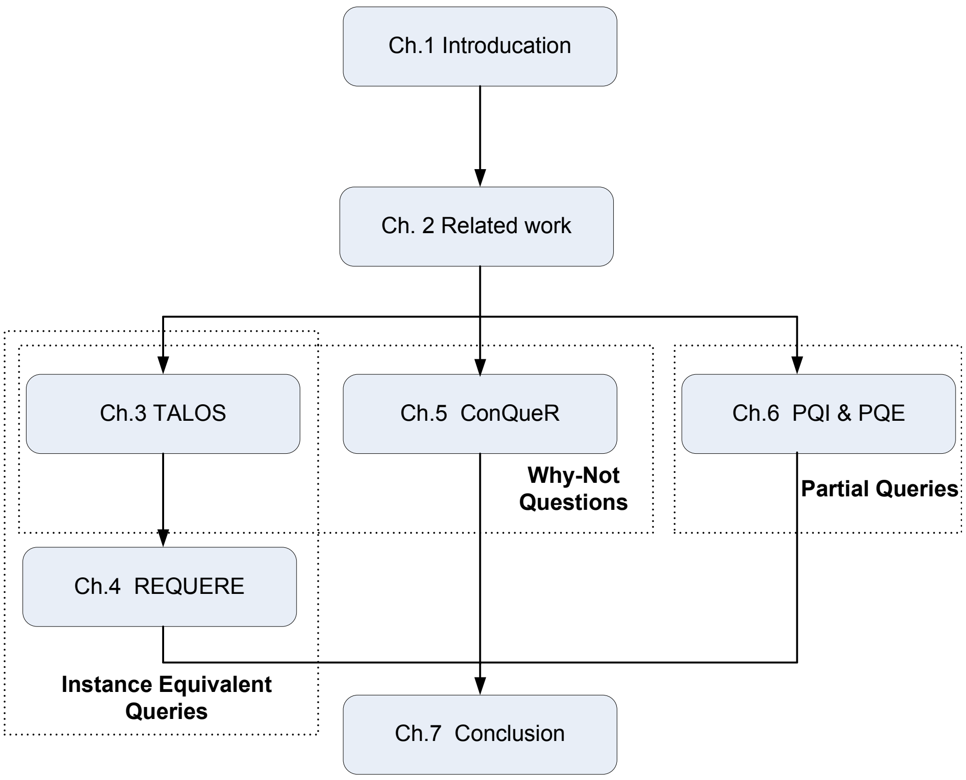


Figure 1-2: Thesis Structure

three parameters of the problem setting including: (1) the original query Q (e.g., Q might not be given), (2) the database version D (e.g., there have multiple versions of the database D), and (3) the derived query Q' (e.g., Q' is in the more expressive fragments with the presence of union/aggregation operator).

- *Explaining Why-not Questions*: Chapter 5 describes our proposed techniques for the second variant of `QBO` to provide a new explanation model with a more flexible constraint-based method, `ConQueR`, for explaining why-not questions. We also discuss how to use `ConQueR` and `TALOS` for explaining why-not questions and deriving instance-equivalent queries in this chapter.

- *Instantiating & Evaluating Partial Queries*: Chapter 6 presents our proposed approaches to evaluate and instantiate partial queries. For `PQE` problem, we introduce two evaluation algorithms, `DP` and `Greedy`. For `PQI`, we propose two approaches, `LA` and $\texttt{LA}^e$, to instantiate partial queries.

- *Conclusion*: Finally, Chapter 7 concludes our work and discusses some interesting directions that future studies can undertake.
- *Appendix*: The proofs of the theoretical studies in our work are given in the Appendix.

Chapter 2

Literature Review

Although the title *Query by Output* (QBO) of our work is inspired by Zloof's influential work on *Query by Example* (QBE) [57], the problem addressed by QBE is completely different from QBO. In particular, QBE receives an input query Q through a graphical user interface that is a more intuitive form-based interface for database querying, and computes its output $Q(D)$ w.r.t. a given database D. In contrast, QBO takes the query result $Q(D)$ of some query Q on a database D as input, and computes a set of queries Q'_1, $\cdots$, Q'_n such that each $Q'_i(D)$ is (approximately) equal to $Q(D)$.

In the following discussions, we classify the related work of QBO in terms of their similarities/differences with the three settings of QBO. In particular, Section 2.1 presents the related work of the first setting of QBO that aims to derive interesting characterizations of tuples in the query result, and Section 2.2 presents the related works that share the same broad principle of "reverse query processing" as QBO. Section 2.3 then discusses existing explanation models to explain why-not questions (the second variant of QBO). Finally, Section 2.4 presents the related work of the third setting of QBO to evaluate and instantiate partial queries.

2.1 Deriving Instance Equivalent Queries

To the best of our knowledge, QBO is the first data-driven approach that aims to augment query results with interesting query-based characterizations of the tuples in the query result. A somewhat related problem of QBO, called *View Definition Problem* (VDP), is introduced later on in [43]. VDP examines the problem of deriving a view definition Q given an input database D and a materialized view V. However, VDP focuses on a very basic scenario, where D consists of a single relation R and the derivation of Q is essentially finding the selection predicate on R to generate V. In addition, since VDP does not handle the *at-least one* semantics that is inherent in the derivation of the IEQs in QBO problem but not in VDP problem, the solutions for VDP cannot be generalized to solve QBO.

An area that is related and complementary to QBO is intensional query answering (IQA) or cooperative answering, where for a given Q, the goal of IQA is to augment the query's answer $Q(D)$ with additional "intensional" information in the form of a semantically equivalent query[1] that is derived from the database integrity constraints [15, 36]. While semantic equivalence is stronger than instance equivalence and can be computed in a data-independent manner using only integrity constraints (ICs), there are several advantages of adopting instance equivalence for QBO. First, in practice, many data semantics are not explicitly captured using ICs in the database for various reasons [17]. For instance, the expense of integrity checking has always limited people's use of ICs, or it may be very hard to justify a useful semantic characterization as an integrity constraint. Hence, the effectiveness of IQA could be limited for QBO. Second, even when the ICs in the database are complete, it can be very difficult to derive semantically equivalent queries for complex queries (e.g., skyline queries that select dominant objects). By being data-specific, IEQs can often provide insightful and surprising characterizations of the input query and its result. Third, IQA requires the input query Q to be known. IQA, therefore, cannot be applied to QBO applications where only $Q(D)$ (but not Q) is

[1]Two queries Q and Q' are *semantically equivalent* if for every valid database D, $Q(D) = Q'(D)$.

available. Thus, we view IQA and our proposed data-driven approach to compute IEQs as complementary techniques for `QBO`.

More recently, an interesting direction of using Précis queries [28, 44] has been proposed. The idea is to augment a user's query result with other related information (e.g., relevant tuples from other relations) and also allow the results to be personalized based on user-specified or domain requirements. The objectives of this work are orthogonal to `QBO`; and as in IQA, it is a query-driven approach that requires the input query to be known.

In the data mining literature, a somewhat related problem to ours is the problem of *redescription mining* introduced by Ramakrishnan [40]. The goal in redescription mining is to find different subsets of data that afford multiple descriptions across multiple vocabularies covering the same data set. At an abstract level, our work is different from these methods in several ways. First, we are concerned with a fixed subset of the data (the output of the query). Second, none of the approaches for redescription mining accounts for structural (relational) information in the data (something we explicitly address). Third, redescription mining, as it was originally posited, requires multiple independent vocabulary descriptions to be identified. We do not have this requirement, as we are simply interested in alternative query formulations within an SQL context. Finally, the notion of *at-least-one* semantics described in our work is something redescription mining is not concerned with, as it is an artifact of the SQL context of our work.

A somewhat related work of our proposed approach to derive instance-equivalent queries (`TALOS`) is the CrossMine approach for multi-relational classification [56]. CrossMine solves the following classification problem: Given a target relation R_t with tuples that have fixed class labels (i.e., positive or negative), build a decision tree classifier for tuples in R_t using the attributes in R_t as well as the attributes from other relations that have primary-foreign key relationships with R_t. `TALOS` differs from CrossMine in several ways. First, there is the notion of *free tuples* in `TALOS`, which are the tuples that can be dynamically assigned positive or negative class labels satisfying some constraints uniquely imposed in `QBO` problem (i.e., at-least-one semantics, exactly-k semantics, aggregation constraints). In contrast, CrossMine does not handle the free tuples due to the nature of the problem solving. The formal definition of free tuples is given in Section 3.1.3. Second, `TALOS` guarantees to find the *optimal* splitting condition at each step of building decision trees whereas the solution of CrossMine is a *greedy* approach.

2.2 Reverse Query Processing

There are several recent works [3, 4, 7, 31, 35] that share the same broad principle of "reverse query processing" as `QBO` but differ totally in the problem objectives and techniques.

The three problems addressed in [3, 4, 31] all aim to generate test databases. In particular, Binnig et al. [3] introduced the *Reverse Query Processing* problem, which receives a query Q and a desired result R to generate a database D such that $Q(D) = R$. Binnig et al. [4] introduced the QAGen problem, which takes as inputs a query Q and a set of target cardinality constraints on intermediate subexpressions in Q's evaluation plan and generates a test database as output such that the evaluation plan of Q on D satisfies the cardinality constraints. A recent work in [31] extends QAGen by replacing the input query Q by a set of workload queries in the inputs. The goal of [31] is to generate a minimal set of database instances such that the workload queries, when executing on these database instances, will satisfy the cardinality constraints. Our `QBO` problem aims to generate instance-equivalent queries and not test databases, which are the goals of these problems.

In another set of related work of `QBO` in terms of reverse query processing principle, Bruno et al. [7] and Mishra et al. [35] examined the problem of *Targeted Query Generation* (TQGen) that aims to generate test queries to meet certain cardinality constraints. TQGen takes as inputs a query Q, a database D, and a set of target cardinality constraints on intermediate subexpressions in Q's evaluation plan. TQGen will modify Q (by modifying the constant values in Q's selection predicates) to generate a new query Q' such that the evaluation plan of Q' on D satisfies the cardinality constraints. Different from TQGen problem, the first setting of `QBO` aims to generate instance-equivalent queries that satisfy the "content constraint" of the query result. In addition, TQGen requires the input query Q to be known whereas `QBO` allows the input query to be unknown. We will clarify the relationship between TQGen and our third setting of `QBO` in Section 2.4.

2.3 Explaining Why-Not Questions

There are currently two existing models in literature to explain why-not questions. The first approach explains by modifying some tuples in the database so that the result of the query on the modified database will include both the original result and the specified missing tuples [22]. This explanation model is very flexible if arbitrary modifications to the database are allowed to derive the missing tuples. However, this model may not be applicable in applications where it may not be meaningful to make arbitrary changes to the stored data. This work is orthogonal to our approach, which is based on modifying the input query. Unlike our work, [22] focuses only on SPJ queries and does not address why-not questions on SPJ queries with aggregation (SPJA queries). A recent work, called Artemis [20, 21], extends [22] by supporting a set of why-not tuples over a set of SPJUA (SPJ with union and/or aggregation operator) queries with constraints among why-not tuples using variables. Our work can also handle SPJUA queries with constraints on why-not tuples.

The second approach, introduced in [9], explains missing tuples by identifying the manipulation operation(s) in the query plan that is responsible for excluding the missing tuples. The work here focuses only on SPJ queries and does not consider SPJA queries. The idea of identifying the "culprit" operator to explain unexpected query results also appears in [14] in the context of explaining mismatches in schema matching. A recent work in [33], which can be categorized in the same class with [9], utilizes the notion of *causality* in Logic to explain why-not questions by also pointing out culprit operators in the query evaluation plans that have eliminated the missing tuples. This model achieves the same goals as [9]; moreover, it can explain both why and why-not questions under the same framework. Our explanation focuses on explaining why-not questions, and goes beyond merely identifying culprit query operator(s) in contrast to [9, 33].

There is also some related work on query refinement to modify an input query so that its query result can satisfy some cardinality constraints. The work in [27, 38] relaxes the *failed* queries that return empty result so that the modified queries will yield some answers. As the goal there is to refine the query to return any non-empty result, the techniques there cannot be applied to our problem, which has stronger constraints to satisfy. Another related direction in [34, 10] deals with the problem when a query returns too many/few answers by refining the query to satisfy some constraint on the query result size. Similar to the work in [27], the focus there is on the size of the output but not on the content of the output, which we have to deal with in this context.

Another related direction is the work on provenance [12] and OLAP [42]. The work in [12] can trace the provenance of an aggregated value by finding the data that derived a given aggregated value. However, the techniques in [12] cannot be extended to handle the why-not questions that we address for SPJA queries. The reason is that in our case for explaining why-not questions for SPJA queries, we need to take into account other data that contributes to produce an aggregated value in addition to the data selected by the original query. The work in [42] addresses explanations for OLAP applications to explain why an aggregated value in a data cube cell is lower/higher than the value in another cell. The main focus there is to compute a compact summarization of the data tuples at the detailed lower levels to account for the phenomena. The work there cannot be generalized to solve our problem related to complex why-not questions on SPJA queries, since we need to take into consideration tuples both in the lower levels and in combination with tuples in the higher and neighbor cells.

Essentially, our proposed framework, `TALOS`, to generate instance-equivalent queries of an input query to solve the first setting of QBO can be applied to derive refined queries to explain why-not questions. Indeed, we have extended TALOS as an alternative solution for this work. Our experimental results reveal that TALOS is a more precision-oriented approach; thus, the queries generated can be rather different from the input query. In some applications, it may not be too meaningful to explain missing tuples using refined queries that are very different from the input query. Moreover, the performance of TALOS is also slower than the proposed approach for explaining why-not questions by up to factor of 6 times due to its costly data classification step.

2.4 Instantiation & Evaluation of Partial Queries

There are two threads of related work corresponding to the two problems of evaluating and instantiating partial queries.

Technique	**Number of constraints**				**Type of optimization constraint**	**Time complexity**
	count	**sum**	**group by count**	**group by sum**		
CourseRank[2]	1	0	0/1	0	unbounded sum	polynomial
Knapsack[26, 18]	0	≥ 1	0	0/1	unbounded sum	PP
Subset sum[26]	0	≥ 1	0	0/1	bounded sum, none	PP
DP(our work)	1	≥ 0	0	0	sum	PP
	0	≥ 0	x	$1-x$	sum	
	0	≥ 0	0	0	count	
Greedy(our work)	≥ 0	≥ 0	x	$1-x$	sum, count, none	heuristic

Table 2.1: Comparison of PQE with respect to sum, count, group by & optimization constraints (PP: pseudo-polynomial)

For PQE, there are three related problem formulations [2, 18, 26] that have been studied and correspond to special cases of partial queries, as shown in Table 2.1. Table 2.1 compares these related works with our proposed evaluation algorithms, DP and Greedy, in terms of evaluating partial queries containing different combinations of sum, count, group by, and optimization constraints. Each row (or collection of rows) describes one technique. Columns 2 to 5 indicate the properties of the partial query fragment considered in terms of the number of count, sum, group by count, and group by sum constraints supported. Column 6 indicates the type of optimization constraints supported in the partial query fragment considered. An optimization constraint is classified as a bounded constraint if it limits the upper value (resp. lower value) of the aggregated function when the condition is "$\leq$" (resp. $\geq$). Otherwise, an optimization constraint that is not bounded is classified as *unbounded*. As an example, the constraint $maximize(sum(A_i)) \leq c$ is a bounded optimization constraint on sum, while the constraint $maximize(sum(A_i))$ is an unbounded optimization constraint on sum. The formal definitions of the constraints supported by PQE are presented in Section 6.1. Finally, the last column indicates the complexity of the proposed approach if it is an optimal algorithm; otherwise, the last column indicates that the proposed approach is a heuristic solution.

The two classic knapsack problem (KP) and subset-sum problem (SSP) [26] correspond to the second and third row in Table 2.1. In KP, given a set of items, where each item j has a profit p_j and a weight w_j, the goal is to select a subset of items such that its total profit is maximized and its total weight does not exceed an input capacity value. A variant of KP was studied in [18] for solving optimization under parametric aggregation constraints (OPAC) query, which takes the following as inputs: (1) a relation $R(A_1, \cdots, A_n, P)$, (2) a set of parametric sum-aggregation constraints of the form $sum(A_i) \leq c_i$ with c_i as a parameter, and (3) a sum optimization constraint $sum(P)$ to be maximized. Given a parameterized OPAC query, [18] proposed an algorithm to construct indices to efficiently provide approximate answers with guarantee bound on its accuracy for any instantiated OPAC query with specific values for the parameters in the sum constraints.

A second variant of KP, which corresponds to the fragment of PQE with multiple sum and a single group-by sum constraints, is the *multiple-choice* Knapsack problem [26]. This is useful in budgeting applications to select a set of projects to be funded such that the total cost for all projects is bounded by some limit, the total cost for projects belonging to the same department is bounded by another limit, and the total project profit is maximized. This problem can be solved in pseudo-polynomial time using a two-step dynamic programming approach [26] which is similar to our proposed DP. However, the formulations of the dynamic programming in each method are different from the other, since DP needs to take into account the count/content constraints, which [26] does not consider.

Another related work is the CourseRank (CR) project [2] which is motivated by course planning applications. CR considers constraints of the form "take at least a and at most b courses from a set S_i", where a and b are non-negative integers and S_i is a set of related courses (e.g. CS courses), and each course is associated with a use-preference score. For example, a student might be required to complete 2 or 3 courses from a given set of six math courses. Given such constraints, CR finds a set of courses that satisfies all requirements such that the number of selected courses is equal to some given value and the total score of the selected courses is maximized. A polynomial-time algorithm based on maximal flow was proposed for the CR problem [2].

A somewhat related work of PQE is the problem introduced in [1], which is motivated by online shopping applications to recommend "satellite items" (e.g., case, speaker) related to a given "center item" (e.g., iPhone). Given a budget B and a central item, [1] finds (approximately) all *maximal* sets of satellite items associated with the central item such that the cost of each maximal set does not exceed the given budget B. Different from [1], we do not consider "maximal set" constraints, which will make the problem of evaluating partial queries even harder. However, partial queries support other constraints (e.g., count, group-by, content) which [1] does not handle.

In summary, although several special cases of partial query evaluations have been studied in different contexts, none of these specialized approaches can be applied to evaluate the general partial queries that our heuristic approach, Greedy, is designed to address.

For PQI, the related work of *Targeted Query Generation* (TQG) problem has been addressed in [7, 35]. Our approach of addressing the problem via partial query instantiation is more general, and therefore flexible as the targeted (i.e. instantiated) queries generated by our approach can involve both modifying the constants of the existing selection predicates as well as adding new selection predicates. This flexibility is important, as it may not be always possible to produce the targeted queries without adding additional predicates.

Chapter 3

TALOS: A Classification-based Approach for Query by Output

In this chapter, we present our classification-based approach, named TALOS, to handle the first setting of *Query by Output* problem to derive instance-equivalent queries (IEQs) for a given input query w.r.t. a given database, highlighted in Table 3.1. We start the discussion with an overview about TALOS in Section 3.1, followed by the techniques of TALOS to handle the *at-least-one* semantics, which is uniquely imposed in QBO, in Section 3.2; and the general framework of TALOS in Section 3.3. We introduce the comparison metrics to rank the returned IEQs in Section 3.4. We describe the implementation details and analyze the complexity of TALOS in Section 3.5. We then conduct experimental studies in Section 3.6 to evaluate the usefulness of TALOS as well as the interestingness of the returned IEQs. Finally, we summarize our work on deriving IEQs in Section 3.7. Part of the contents and materials in this chapter were previously published in [49].

3.1 An Overview of TALOS

The QBO problem takes as inputs a database D, an optional query Q, and the query's output $Q(D)$ (w.r.t. D) to compute one or more IEQs Q', where Q and Q' are IEQs if

QBO Problem	Parameters	
	Input query Q	Given result table T
The first variant	**- Q is known** - Q is unknown	$\mathbf{T = Q(D)}$ T is a set of specific tuples
The second variant	Q is known	$T = Q(D) \bigcup S$ S is a set of tuples that are not present in $Q(D)$
The third variant	Q is partially specified	T is a set of constraints that must be satisfied by the query result of each derived query Q'

Table 3.1: The Focus of Chapter 3

$Q(D) = Q'(D)$. We refer to Q as the *input query*, $Q(D)$ as the *given result table*, and Q' as the *output query*.

First, let us state the following theoretical results that we have established for variants of the QBO problem.

Theorem 3.1 *Given an input query Q, we define QBO_S to be the problem to find an output query Q' w.r.t. a database D, where Q' involves only selection (with predicates in the form "A_i op c", A_i is an attribute, c is constant, and $op \in \{<, \leq, =, \neq, >, \geq\}$) such that: (1) $Q'(D) = Q(D)$, and (2) the number of operators (AND, OR and NOT) used in the selection condition is not greater than a given constant s. Then, QBO_S is believed not to be in P.*

Proof Sketch: We prove Theorem 3.1 by reducing the Minimization Circuit Size Problem to QBO_S. Details are given in Appendix A. □

Theorem 3.2 *Given an input query Q, we define QBO_U to be the problem to find an output query Q' w.r.t. a database D in the form $Q' = Q_1$ union $\cdots$ union Q_k, with each Q_i is an SPJ query and the select-clause of Q_i refers to only attributes of relations in Q_i, such that: (1) $Q'(D) = Q(D)$, and (2) k is not greater than a given constant n. Then QBO_U is NP-hard.*

Proof Sketch: We prove Theorem 3.2 by reducing the Set-Covering to QBO_U. Details are given in Appendix B. □

Theorem 3.3 *Given an input query Q, we define QBO_G to be the problem to find an output query Q' w.r.t. a database D such that: (1) $Q'(D) = Q(D)$, and (2) users can specify any constraints on the clauses of Q' (e.g., the select clause of Q' can contain arbitrary arithmetic expressions or the where-clause of Q' must contain some specific selection conditions). Then QBO_G is PSPACE-hard.*

Proof Sketch: We prove Theorem 3.3 by reducing the Integer Circuit Evaluation Problem to QBO_G. Details are given in Appendix C. □

In this work, we consider relational queries Q where the select-clause refers to only attributes (and not to constants or arithmetic/string expressions) to ensure that Q' can be derived from $Q(D)$ efficiently. We also require that $Q(D) \neq \emptyset$ for the problem to be interesting.

For simplicity, we first consider Select-Project-Join (SPJ) queries for the IEQ Q' where all the join predicates in Q' are foreign-key joins. Thus, our approach requires only very basic database integrity constraint information (i.e., primary and foreign key constraints). Based on the knowledge of the primary and foreign key constraints in the database, the database schema can be modeled as a *schema graph*, denoted by $\mathcal{SG}$. Each node in $\mathcal{SG}$ represents a relation, and each edge between two nodes represents a foreign-key join between the relations corresponding to the nodes. We defer the discussions on finding IEQs in more expressive fragments (e.g., SPJ queries with union or aggregation operations) to Chapter 4.

For ease of presentation and without loss of generality, we express each Q' as a relational algebra expression. To keep our definition and notations simple and without loss of generality, we shall assume that there are no multiple instances of a relation in Q and Q'.

Running example. We use a database housing baseball statistics for our running example as well as in our experiments. Part of the schema is illustrated in Figure 3-1, where the key attribute names are shown in bold. The *Master* relation describes information

pID	name	country	weight	bats	throws
P1	A	USA	85	L	R
P2	B	USA	72	R	R
P3	C	USA	80	R	L
P4	D	Germany	72	L	R
P5	E	Japan	72	R	R

(a) *Master*

pID	**year**	salary
P1	2003	80
P3	2002	35
P5	2004	60

(b) *Salaries*

pID	**year**	**stint**	team	HR
P1	2001	2	PIT	40
P1	2003	2	ML1	50
P2	2001	1	PIT	73
P2	2002	1	PIT	40
P3	2004	2	CHA	35
P4	2001	3	PIT	30
P5	2004	3	CHA	60

(c) *Batting*

team	**year**	rank
PIT	2001	7
PIT	2002	4
CHA	2004	3

(d) *Team*

Figure 3-1: Running Example - Baseball Data Set *D*

about each player (identified by *pID*): the attributes *name*, *country*, *weight*, *bats*, and *throws* refer to his name, birth country, weight (in pounds), batting hand (left, right, or both), and throwing hand (left or right) respectively. The *Salaries* relation specifies the salary obtained by a player in a specific year. The *Batting* relation provides the number of home runs (*HR*) of a player when he was playing for a team in a specific year and season (*stint*). The *Team* relation specifies the rank obtained by a team for a specified year.

Notations. Given a query Q, we use *rel(Q)* to denote the collection of relations involved in Q (i.e., relations in SQL's from-clause); *proj(Q)* to denote the set of projected attributes in Q (i.e., attributes in SQL's select-clause); and *sel(Q)* to denote the set of selection predicates in Q (i.e., conditions in SQL's where-clause).

3.1.1 Instance-Equivalent Queries (IEQs)

Our basic definition of *instance-equivalent queries (IEQs)* requires that the IEQs Q and Q' produce the same output (w.r.t. some database D); i.e., $Q(D) = Q'(D)$. The advantage of this simple definition is that it does not require the knowledge of Q to derive Q', which is particularly useful for `QBO` applications where Q is either missing or not provided. However, there is a potential "accuracy" tradeoff that arises from the simplicity of this weak form of equivalence: an IEQ may be "semantically" quite different from the input query that produced $Q(D)$ as the following example illustrates.

Example 3.1 *Consider the following three queries on the baseball database D in Figure 3-1:*

$Q_1 = \pi_{country}(\sigma_{bats=\text{“}R\text{”} \wedge throws=\text{“}R\text{”}}(Master)),$

$Q_2 = \pi_{country}(\sigma_{bats=\text{“}R\text{”} \wedge weight \leq 72}(Master)),$ *and*

$Q_3 = \pi_{country}(\sigma_{bats=\text{“}R\text{”}}(Master)).$

Observe that although all three queries produce the same output after projection ({USA, Japan}), only Q_1 and Q_2 select the same set of tuples $\{P2, P5\}$ from Master. Specifically, if we modify the queries by replacing the projection attribute "country" with the key attribute "pID", we have $Q_1(D) = \{P2, P5\}$, $Q_2(D) = \{P2, P5\}$, and $Q_3(D) = \{P2, P3, P5\}$. Thus, while all three queries are IEQs, we see that the equivalence between Q_1 and Q_2 is actually "stronger" (compared to that between Q_1 and Q_3) in that both queries actually select the same set of relation tuples. □

If Q is provided as part of the input, then we can define a stronger form of instance equivalence as suggested by the above example. Intuitively, the stricter form of instance equivalence not only ensures that the instance-equivalent queries produce the same output (w.r.t. some database D), but it also requires that their outputs be projected from the same set of "core" tuples. We now formally characterize weak and strong IEQs based on the concepts of *core relations* and *core queries*.

Core relations. Given a query Q, we say that $S \subseteq rel(Q)$ is a set of *core relations* of Q if S is a minimal set of relations such that for every attribute $R_i.A \in proj(Q)$: (1) $R_i \in S$, or (2) Q contains a chain of equality join predicates "$R_i.A = \cdots = R_j.B$" such that $R_j \in S$.

Intuitively, a set of core relations of Q is a minimal set of relations in Q that "cover" all the projected attributes in Q. As an example, if $Q = \pi_{R_1.X}\sigma_p(R_1 \times R_2 \times R_3)$ where $p = (R_1.X = R_3.Y) \wedge (R_2.Z = R_3.Z)$, then Q has two sets of core relations, $\{R_1\}$ and $\{R_3\}$.

Core queries. Given a query Q and a set of relations $S \subseteq rel(Q)$, we use Q_S to denote the query that is derived from Q by replacing $proj(Q)$ with the key attribute(s) of each relation in S. If S is a set of core relations of Q, we refer to Q_S as a *core query* of Q.

Strong & weak IEQs. Consider two IEQs Q and Q' (w.r.t. a database D); i.e., $Q(D) = Q'(D)$. We say that Q and Q' are *strong IEQs* if Q has a set of core relations S such that: (1) Q'_S is a core query of Q', and (2) $Q_S(D)$ and $Q'_S(D)$ are equal. IEQs that are not strong are classified as *weak IEQs*.

The strong IEQ definition essentially requires that both Q and Q' share a set of core relations such that $Q(D)$ and $Q'(D)$ are projected from the same set of selected tuples from these core relations. Thus, in Example 3.1, Q_1 and Q_2 are strong IEQs whereas Q_1 and Q_3 are weak IEQs.

Note that in our definition of strong IEQ, we only impose moderate restrictions on Q and Q' (relative to the weak IEQ definition) so that the space of strong IEQs is not overly constrained, and that the strong IEQs generated are hopefully both interesting as well as meaningful.

As in the case with weak IEQs, two strong IEQs can involve different sets of relations. As an example, suppose query Q selects pairs of records from two core relations, *Supplier* and *Part*, that are related via joining with a (non-core) *Supply* relation. Then it is possible for a strong IEQ Q' to relate the same pair of core relations via a different relationship (e.g., by joining with a different non-core *Manufacture* relation).

We believe that each of the various notions of query equivalence has useful applications in different contexts depending on the available type of information about the input query and database. At one extreme, if both Q and the database integrity constraints are available, we can compute both weak and strong IEQs. At the other extreme, if only $Q(D)$ and the database D are available, we can only compute weak IEQs.

Precise & approximate IEQs. It is also useful to permit some perturbation so as to include IEQs that are "close enough" to the original. Perturbations could be in the form of extra records or missing records or a combination thereof. Such generalizations are necessary in situations where there are no precise IEQs, and useful for cases where the computational cost for finding precise IEQs is considered unacceptably high. Moreover, a precise IEQ Q' might not always provide insightful characterizations of $Q(D)$, as Q' could be too "detailed" with many join relations and/or selection predicates.

The imprecision of a weak IEQ Q' of Q (w.r.t. D) can be quantified by $|Q(D) - Q'(D)| + |Q'(D) - Q(D)|$; the imprecision of a strong IEQ can be quantified similarly. Thus, Q' is considered an approximate (strong/weak) IEQ of Q if its imprecision is positive; otherwise, Q' is a precise (strong/weak) IEQ.

3.1.2 TALOS: Conceptual Approach

In this section, we give a conceptual overview of our approach, named `TALOS` (for **T**ree-based classifier with **A**t **L**east **O**ne **S**emantics), for the `QBO` problem.

Given a given result table $Q(D)$, to generate an SPJ Q' that is an IEQ of Q, we basically need to determine the three components of Q': $rel(Q')$, $sel(Q')$, and $proj(Q')$. Clearly, if $rel(Q')$ contains a set of core relations of Q, then $proj(Q')$ can be trivially derived from these core relations[1]. Thus, the possibilities for Q' depend mainly on the options for both $rel(Q')$ and $sel(Q')$. Between these two components, enumerating different $rel(Q')$ is the easier task, as $rel(Q')$ can be obtained by choosing a subgraph G of the schema graph $\mathcal{SG}$ such that G contains a set of core relations of Q: $rel(Q')$ is then given by all the relations represented in G. Note that it is not necessary for $rel(Q) \subseteq rel(Q')$, as Q may contain some relations that are not core relations. The reason for exploring different possibilities for $rel(Q')$ is to find interesting alternative characterizations of $Q(D)$ that involve different join paths or selection conditions from those in Q. `TALOS` enumer-

[1]Note that even though the definition of a weak IEQ Q' of Q does not require the queries to share a set of core relations, we find this restriction to be a reasonable and effective way to obtain "good" IEQs.

ates different schema subgraphs by starting out with minimal subgraphs that contain a set of core relations of Q, and then incrementally expanding the minimal subgraphs to generate larger, more complex subgraphs.

We now come to the most critical and challenging part of our solution, which is how to generate "good" $sel(Q')$'s such that each $sel(Q')$ is not only succinct (without too many conditions) and insightful, but also minimizes the imprecision between $Q(D)$ and $Q'(D)$ if Q' is an approximate IEQ. We propose to formulate this problem as a *data classification* task as follows.

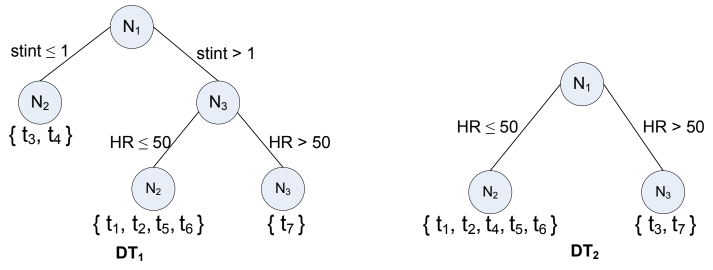
Consider the relation J that is computed by joining all the relations in $rel(Q')$ based on the foreign-key joins represented in G. Without loss of generality, let us suppose that we are looking for weak IEQs Q'. Let L denote the ordered listing of the attributes in $proj(Q')$ such that that the schema of $\pi_L(J)$ and $Q(D)$ are equivalent[2]. J can be partitioned into two disjoint subsets, $J = J_0 \cup J_1$, such that $\pi_L(J_1) \subseteq Q(D)$ and $\pi_L(J_0) \cap Q(D) = \emptyset$. For the purpose of deriving $sel(Q')$, one simple approach to classify the tuples in J is to label the tuples in J_0, which do not contribute to the query's result $Q(D)$, as *negative tuples*, and label the tuples in J_1 as *positive tuples*.

Given the labeled tuples in J, the problem of finding a $sel(Q')$ can now be viewed as a data classification task to separate the positive and negative tuples in J: $sel(Q')$ is given by the selection conditions that specify the positive tuples. A natural solution is to examine if off-the-shelf data classifier can give us what we need. To determine what kind of classifier to use, we must consider what we need to generate our desired IEQ Q'. Clearly, the classifier should be efficient to construct and the output should be easy to interpret and express using SQL; i.e., the output should be expressible in axis parallel cuts of the data space. These criteria rule out a number of classifier systems such as neural networks, k-nearest neighbor classification, Bayesian classifiers, and support vector machines [46]. Rule based classifiers or decision trees (a form of rule-based classifier) are a natural solution in this context. `TALOS` uses decision tree classifier for

[2]If the search is for strong IEQs, then the discussion remains the same except that L is the ordered listing of the key attributes of a set of core relations S of Q, and we replace $Q(D)$ by $Q_S(D)$.

	pID	name	country	weight	bats	throws	year	team	stint	HR
t_1	P1	A	USA	85	L	R	2001	PIT	2	40
t_2	P1	A	USA	85	L	R	2003	ML1	2	50
t_3	P2	B	USA	72	R	R	2001	PIT	1	73
t_4	P2	B	USA	72	R	R	2002	PIT	1	40
t_5	P3	C	USA	80	R	L	2004	CHA	2	35
t_6	P4	D	Germany	72	L	R	2001	PIT	3	30
t_7	P5	E	Japan	72	R	R	2004	CHA	3	60

(a) $J = Master \bowtie_{pID} Batting$



(b) Decision trees DT_1 and DT_2

Figure 3-2: Example of deriving IEQs for Q_4 on D

generating $sel(Q')$.

We now briefly describe how a simple binary decision tree is constructed to classify a set of data records D. For expository simplicity, assume that all the attributes in D have numerical domains. A decision tree DT is constructed in a top-down manner. Each leaf node N in the tree is associated with a subset of the data records, denoted by D_N, such that D is partitioned among all the leaf nodes. Initially, DT has only a single leaf node (i.e., its root node), which is associated with all the records in D. Leaf nodes are classified into pure and non-pure nodes depending on a given goodness criterion. Common goodness criteria include entropy, classification error and the Gini index [46]. At each iteration of the algorithm, the algorithm examines each non-pure leaf node N and computes the best split for N that creates two child nodes, N_1 and N_2, for N. Each split is computed as a function of an attribute A and a split value v associated with the attribute. Whenever a node N is split (w.r.t. attribute A and split value v), the records in D_N are partitioned between D_{N_1} and D_{N_2} such that a tuple $t \in D_N$ is distributed into D_{N_1} if $t.A \leq v$; and D_{N_2}, otherwise.

A popular goodness criterion for splitting, the Gini index, is computed as follows. For a data set S with k distinct classes, its Gini index is $Gini(S) = 1 - \sum_{j=1}^{k}(f_j^2)$, where f_j denotes the fraction of records in S belonging to class j. Thus, if S is split into two subsets S_1 and S_2, then the Gini index of the split is given by

$$Gini(S_1, S_2) = \frac{|S_1|\, Gini(S_1) + |S_2|\, Gini(S_2)}{|S_1| + |S_2|},$$

where $|S_i|$ denotes the number of records in S_i. The general objective is to pick the splitting attribute whose best splitting value reduces the Gini index the most (the goal is to reduce Gini to 0 resulting in all pure leaf nodes).

Example 3.2 *To illustrate how decision tree classifier can be applied to derive IEQs, consider the following query on the baseball database D:* $Q_4 = \pi_{name}\,(\sigma_{bats=\text{“R”} \wedge throws=\text{“R”}}$ $Master)$. *Note that* $Q_4(D) = \{B, E\}$. *Suppose that the schema subgraph G considered contains both Master and Batting; i.e.,* $rel(Q_4') = \{Master, Batting\}$. *The output of* $J =$ $Master \bowtie_{pID} Batting$ *is shown in Figure 3-2(a). Using* t_i *to denote the* i^{th} *tuple in J, we observe that J is partitioned into* $J_0 = \{t_1, t_2, t_5, t_6\}$ *and* $J_1 = \{t_3, t_4, t_7\}$. *Figure 3-2(b) shows two example decision trees,* DT_1 *and* DT_2, *constructed from J. Each decision tree partitions the tuples in J into different subsets (represented by the leaf nodes) by applying different sequences of attribute selection conditions. By labeling all tuples in* J_1 *as positive, the IEQ derived from* DT_1 *is given by* $Q_4' = \pi_{name}(\sigma_{stint \leq 1 \vee (stint > 1 \wedge HR > 50)}$ $(Master \bowtie Batting))$. □

3.1.3 TALOS: Challenges

There are two key challenges in adapting decision tree classifier for the `QBO` problem.

At Least One Semantics. The first challenge concerns the issue of how to assign class labels in a flexible manner without over constraining the classification problem and limiting its effectiveness. Contrary to the impression given by the above simple class labeling scheme, the task of assigning class labels to J is actually a rather intricate problem due to the fact that multiple tuples in J_1 can be projected to the same tuple in $\pi_L(J_1)$. Recall that in the simple class labeling scheme described, a tuple t is labeled positive if and only if $t \in J_1$. However, note that it is possible to label only a subset of tuples $J_1' \subseteq J_1$ as positive (with tuples in $J - J_1'$ labeled as negative), and yet achieve $\pi_L(J_1') = \pi_L(J_1)$ (without affecting the imprecision of Q'). In other words, the simple scheme of labeling all tuples in J_1 as positive is just one (extreme) option out of many other possibilities.

We now discuss more precisely the various possibilities of labeling positive tuples in J to derive different $sel(Q')$. Let $\pi_L(J_1) = \{t_1, \cdots, t_k\}$. Then J_1 can be partitioned into k subsets, $J_1 = P_1 \cup \cdots \cup P_k$, where each $P_i = \{t \in J_1 \mid$ the projection of t on L is $t_i\}$. Thus, each P_i represents the subset of tuples in J_1 that project to the same tuple in $\pi_L(J_1)$. Define J_1' to be a subset of tuples of J_1 such that it consists of at least one tuple from each subset P_i. Clearly, $\pi_L(J_1') = \pi_L(J_1)$, and there is a total of $\prod_{i=1}^{k}(2^{|P_i|} - 1)$ possibilities for J_1'. For a given J_1', we can derive $sel(Q')$ using a data classifier based on labeling the tuples in J_1' as positive and the remaining tuples in $J_1 - J_1'$ as negative.

Based on the above discussion on labeling tuples, each tuple in J can be classified as either a *bound tuple* or *free tuple* depending on whether there is any freedom to label the tuple. A tuple $t \in J$ is a *bound tuple* if either (1) $t \in J_0$, in which case t must be labeled negative, or (2) t is the only tuple in some subset P_i, in which case t must certainly be included in J_1' and be labeled positive. Otherwise, t is a *free tuple*; i.e., t is in some subset P_i that contains more than one tuple.

In contrast to the conventional classification problem where each record in the input data comes with a well defined class label, the classification problem formulated for QBO has the unique characteristic where there is some flexibility in the class label assignment. We refer to this property as *at-least-one semantics*. In the scenarios when there is a constraint on the number of instances of some specific tuple in the query result (e.g., when there is no "distinct" keyword in the select-clause), the at-least-one semantics becomes *exactly-k semantics*. More specifically, assume there are k instances of a tuple t_i in the query result $Q(D)$, the *exactly-k semantics* requires that J_1' must contain exactly k tuples from the subset $J_i \subseteq J$ corresponding to t_i. To the best of our knowledge, we are not aware of any work that has addressed this variant of the classification problem. For simplicity and without loss of generality, we will mainly focus on the at-least-one semantics in the following discussion. We defer the discussion about the exactly-k semantics to Section 3.2.2.

An obvious approach to solve the at-least-one semantics variant is to map the problem into the traditional variant by first applying some arbitrary class label assignment that is consistent with the at-least-one semantics. In our experimental study, we compare against two such static labeling schemes, namely, (1) `NI`, which labels all free tuples as positive, and (2) `RD`, which labels a random non-empty subset of free tuples in each P_i as positive[3]. However, such static labeling schemes do not exploit the flexible class labeling opportunities to optimize the classification task. To avoid the limitations of the static scheme, `TALOS` employs a novel *dynamic class labeling scheme* to compute optimal node splits for decision tree construction without having to enumerate an exponential number of combinations of class labeling schemes for the free tuples.

Example 3.3 *Continuing with Example 3.2, J_1 is partitioned into two subsets: $P_1 = \{t_3, t_4\}$ and $P_2 = \{t_7\}$, where P_1 and P_2 contribute to the outputs "B" and "E", respectively. The tuples in J_0 and P_2 are bound tuples, while the tuples in P_1 are free tuples. To derive an IEQ, at least one of the free tuples in P_1 must be labeled positive. If t_3 is labeled positive and t_4 is labeled negative, DT_2 in Figure 3-2(b) is a simpler decision tree constructed by partitioning J based on a selection predicate on attribute HR. The IEQ derived from DT_2 is $Q_4'' = \pi_{name}\, \sigma_{HR>50}\, (Master \bowtie Batting)$.* □

Performance Issues. The second challenge concerns the performance issue of how to efficiently generate candidates for $rel(Q')$ and optimize the computation of the single input table J required for the classification task. To improve performance, `TALOS` exploits

[3]We also experimented with a scheme that randomly labels only one free tuple for each subset as positive, but the results are worse than `NI` and `RD`.

	Number of free tuples to be labeled positive		Labeling of free tuples		Exactly-One Constraint Propagation	
Case	f_1	f_2	positive	negative	S_1	S_2
$C1$	$\sum_{i=1}^{m} n_{i,1}$	$\sum_{i=1}^{m} n_{i,2}$	$S_1 \cup S_2$	-	-	-
$C2$	$\sum_{i=1}^{m} n_{i,1}$	T_2	S_1	SP_{12}-sets in S_2	-	SP_2-sets
$C3$	T_1	$\sum_{i=1}^{m} n_{i,2}$	S_2	SP_{12}-sets in S_1	SP_1-sets	-
$C4$	T_1	$m - T_1$	-	SP_{12}-sets in S_1	SP_1-sets	All subsets
$C5$	$m - T_2$	T_2	-	SP_{12}-sets in S_2	All subsets	SP_2-sets

Table 3.2: Optimizing Node Splits

join indices to avoid a costly explicit computation of J, and constructs mapping tables to optimize decision tree construction.

3.2 Handling At-Least-One Semantics

In this section, we address the first challenge of TALOS and present a novel approach for classifying data with the at-least-one semantics. At the end of Section 3.2.2, we will discuss how to adapt the technique of TALOS for at-least-one semantics to handle the exactly-k semantics.

3.2.1 Computing Optimal Node Splits

The main challenge for classification with the at-least-one semantics is how to optimize the node splits given the presence of free tuples that offers flexibility in the class label assignment. We present a novel approach that computes the optimal node split *without* having to explicitly enumerate all possible class label assignments to the free tuples. The idea is based on exploiting the flexibility offered by the at-least-one semantics.

Let us consider an initial set of tuples S that has been split into two subsets, S_1 and S_2, based on a value v of a numeric attribute A (the same principle applies to categorical attributes as well); i.e., a tuple $t \in S$ belongs to S_1 iff $t.A \leq v$. The key question is how to compute the optimal Gini index of this split without having to enumerate all possible class label assignments for the free tuples in S such that the at-least-one semantics is satisfied. Without loss of generality, suppose that the set of free tuples in S is partitioned (as described in Section 3.1.3) into m subsets, $P_1, \cdots, P_m$, where each $|P_i| > 1$.

Let $n_{i,j}$ denote the number of tuples in $P_i \cap S_j$, and f_j denote the number of free tuples in S_j to be labeled positive to minimize $Gini(S_1, S_2)$, where $i \in [1, m]$, $j \in \{1, 2\}$. We classify P_i, $i \in [1, m]$, as a SP_1-*set* (resp. SP_2-*set*) if P_i is completely contained in S_1 (resp. S_2); otherwise, P_i is a SP_{12}-*set* (i.e., $n_{i,1} > 0$ and $n_{i,2} > 0$).

To satisfy the at-least-one semantics, we need to ensure that at least one free tuple in each P_i, $i \in [1, m]$, is labeled positive. Let T_j, $j \in \{1, 2\}$, denote the minimum number of free tuples in S_j that must be labeled positive to ensure this. Observe that for a specific P_i, $i \in [1, m]$, if P_i is a SP_1-set (resp. SP_2-set), then we must have $T_1 \geq 1$ (resp. $T_2 \geq 1$). Thus, T_j is equal to the number of SP_j-sets. More precisely, $T_j = \sum_{i=1}^{m} \max\{0, 1 - n_{i,3-j}\}$, $j \in \{1, 2\}$.

Thus, f_1 and f_2 must satisfy the following two conditions:

(A1) $T_j \leq f_j \leq \sum_{i=1}^{m} n_{i,j}$, $j \in \{1, 2\}$; and

(A2) $f_1 + f_2 \geq m$.

Condition (A1) specifies the possible number of free tuples to be labeled positive for each S_j, while condition (A2) specifies the minimum combined number of tuples in S to be labeled positive in order that the at-least-one semantics is satisfied for each P_i.

Based on conditions (A1) and (A2), it can be shown that the optimal value of $Gini(S_1, S_2)$ can be determined by considering only five combinations of f_1 and f_2 values as indicated by the second and third columns in Table 3.2. The proof of this result is given in Appendix D.

These five cases correspond to different combinations of whether the number of positive or negative tuples is being maximized in each of S_1 and S_2. Case C1 maximizes the number of positive tuples in both S_1 and S_2. Case C2 maximizes the number of positive tuples in S_1 and maximizes the number of negative tuples in S_2. Case C3 maximizes the number of negative tuples in S_1 and maximizes the number of positive tuples in S_2. Finally, cases C4 and C5 maximize the number of negative tuples in both S_1 and S_2. The optimal value of $Gini(S_1, S_2)$ is given by the minimum of the Gini index value derived from the above five cases.

3.2.2 Updating Labels & Propagating Constraints

Once the optimal $Gini(S_1, S_2)$ is determined for a given node split, we need to update the split of S by converting the free tuples in S_1 and S_2 to bound tuples with either positive/negative class labels. The details of this updating depend on which of the five cases the optimal Gini value was derived from, and are summarized by the last four columns in Table 3.2.

For case C1, which is the simplest case, all the free tuples in S_1 and S_2 will be converted to positive tuples. However, for the remaining cases, which involve maximizing the number of negative tuples in S_1 or S_2, some of the free tuples may not be converted to bound tuples. Instead, the maximization of negative tuples in S_1 or S_2 is achieved by propagating another type of constraints, referred to as "exactly-one" constraints, to some subsets of tuples in S_1 or S_2. Similar to the principle of at-least-one constraints, the idea here is to make use of constraints to optimize the Gini index values for subsequent node splits without having to explicitly enumerate all possible class label assignments. Thus, in Table 3.2, the fourth and fifth columns specify which free tuples are to be converted to bound tuples with positive and negative labels, respectively; where an '-' entry means that no free tuples are to be converted to bound tuples. The sixth and seventh columns specify what subsets of tuples in S_1 and S_2, respectively, are required to satisfy the exactly-one constraint; where an '-' entry column means that no constraints are propagated to S_1 or S_2.

We now define the exactly-one constraint and explain why it is necessary. An *exactly-one constraint* on a set of free tuples S' requires that exactly one free tuple in S' must become labeled as positive with the remaining free tuples in S' labeled as negative.

Consider case C2, which is to maximize the number of positive (resp. negative) tuples in S_1 (resp. S_2). The maximization of the number of positive tuples in S_1 is easy to achieve by converting all the free tuples in S_1 to positive, the at-least-one constraints on the SP_1-sets and SP_{12}-sets are also satisfied. Consequently, for each SP_{12}-set P_i, all the free tuples in $P_i \cap S_2$ can be converted to negative tuples (to maximize the number of negative tuples in S_2) without violating the at-least-one constraint on P_i. However, for a SP_2-set P_i, to maximize the number of negative tuples in P_i while satisfying the at-least-one semantics translates to an exactly-one constraint on P_i. Thus, for case C2, an exactly-one constraint is propagated to each SP_2-set in S_2, and no constraints is propagated to S_1. A similar reasoning applies to cases C3 to C5.

Therefore, while the at-least-one constraint is applied to each subset of free tuples P_i in the initial node split, the exactly-one constraint is applied to each P_i for subsequent node splits. This second variant of the node split problem can be optimized by techniques similar to what we have explained so far for the first variant. In particular, the first condition (A1) for f_1 and f_2 remains unchanged, but the second condition (A2) becomes $f_1 + f_2 = m$. Consequently, the optimization of the Gini index value becomes simpler and only needs to consider cases C4 and C5.

Example 3.4 *To illustrate how class labels are updated and how constraints are propagated during a node split, consider the following query on the baseball database D:* $Q_5 = \pi_{stint}$ ($\sigma_{country=\text{“USA”}}$ $Master \bowtie_{pID} Batting$). *Suppose that the weak-IEQ* Q'_5 *being considered has* $rel(Q'_5) = \{Master, Batting\}$. *Let* $J = Master \bowtie_{pID} Batting$ *(shown in Figure 3-2(a)). Since* $Q_5(D) = \{1, 2\}$, *we have* $J_0 = \{t_6, t_7\}$, $P_1 = \{t_1, t_2, t_5\}$ *(corresponding to stint* $= 2$*), and* $P_2 = \{t_3, t_4\}$ *(corresponding to stint* $= 1$*). The tuples in* J_0 *are labeled negative, while the tuples in* P_1 *and* P_2 *are all free tuples.*

Suppose that the splitting attribute considered is “weight”, and the optimal splitting value for “weight” is 72*. The* $Gini(S_1, S_2)$ *values computed (w.r.t. “weight* ≤ 72*”) for the five cases, C1 to C5, are* 0.29, 0.48, 0.21, 0.4 *and* 0.4*, respectively. Thus, the optimal value of* $Gini(S_1, S_2)$ *is* 0.21 *(due to case C3). We then split tuples with weight* ≤ 72

(i.e., $\{t_3, t_4, t_6, t_7\}$) into S_1, and tuples with weight > 72 (i.e., $\{t_1, t_2, t_5\}$) into S_2. Thus, P_1 is a SP_2-set while P_2 is a SP_1-set. Since the optimal Gini index computed is due to case C3 (i.e., maximizing negative tuples in S_1 and maximizing positive tuples in S_2), all the free tuples in S_2 (i.e., t_1, t_2 and t_5) are labeled positive, and an exactly-one constraint is propagated to the set of tuples $P_2 \cap S_1$ (i.e., $\{t_3, t_4\}$). □

Handling Exactly-*k* Semantics. In the following, we discuss how to extend the above technique to solve the *exactly-k* semantics. The exactly-k semantics is required when there is a constraint on the number of instances of some specific tuple in the query result. The *exactly-k* semantics applied on a set of free tuples S' requires that exactly k free tuples in S' must become labeled as positive, while the remaining free tuples in S' are labeled negative. To simplify the presentation, assume that the exactly-k semantics is required on every set of $P_1, \cdots, P_m$.

This variant of the node split problem can be optimized by techniques similar to what we have explained so far for the at-least-one semantic. In particular, the first condition (A1) for f_1 and f_2 becomes $T'_j \leq f_j \leq \sum_{i=1}^{m} n_{i,j}$, $j \in \{1, 2\}$; where $T'_j = \sum_{i=1}^{m} \max\{0, k - n_{i,3-j}\}$, $j \in \{1, 2\}$. The second condition (A2) becomes $f_1 + f_2 = mk$. Consequently, the optimization of the Gini index value becomes simpler and only needs to consider cases C'4 and C'5; where (C'4): $f_1 = T'_1$, $f_2 = mk - T'_1$, and (C'5): $f_1 = mk - T'_2$, $f_2 = T'_2$.

In summary, `TALOS` is able to efficiently compute the optimal Gini index value for each attribute split value considered without enumerating an exponential number of class label assignments for the free tuples.

3.3 TALOS Framework

In this section, we first explain how `TALOS` adapts a well-known decision tree classifier for performing data classification in the presence of free tuples where their class labels are not fixed. We then explain the performance challenges of deriving Q' when $rel(Q')$ involves multiple relations, and present optimization techniques to address these issues.

val	*row*
A	1
B	2
C	3
D	4
E	5

(a) AL_{name}

r_M	r_B	r_T
1	1	1
2	3	1
2	4	2
3	5	3
4	6	1
5	7	3

(b) J_{hub}

r_M	S_{r_J}
1	$\{1\}$
2	$\{2, 3\}$
3	$\{4\}$
4	$\{5\}$
5	$\{6\}$

(c) M_{Master}

nid	*cid*	*sid*
1	0	0
1	-1	1
1	-1	1
1	0	0
1	0	0
1	1	2

(d) CL

Figure 3-3: Example data structures for $Q_4(D)$

For ease of presentation and without loss of generality, the discussion here assumes weak IEQs.

3.3.1 Classifying Data in TALOS

We first give an overview of SLIQ [32], a well-known decision tree classifier, that we have chosen to adapt for TALOS. We then describe the extensions required by TALOS to handle data classification in the presence of free tuples. Finally, we present a non-optimized, naive variant of TALOS. It is important to emphasize that our approach is orthogonal to the choice of the decision tree technique.

Overview of SLIQ. To optimize the decision tree construction on a set of data records D, SLIQ uses two key data structures. First, a sorted attribute list, denoted by AL_i, is pre-computed for each attribute A_i in D. Each AL_i can be thought of as a two-column table (val, row), of the same cardinality as D, that is sorted in non-descending order of val. Each record $r = (v, i)$ in AL_i corresponds to the i^{th} tuple t in D, and $v = t.A_i$. The sorted attribute lists are used to speed up the computation of optimal node splits. To determine the optimal node split w.r.t. A_i requires a single sequential scan of AL_i.

Second, a main-memory array called *class list*, denoted by CL, is maintained for D. This is a two-column table (nid, cid) with one record per tuple in D. The i^{th} entry in CL, denoted by $CL[i]$, corresponds to the i^{th} tuple t in D, where $CL[i].nid$ is the identifier of leaf node N, $t \in D_N$, and $CL[i].cid$ refers to the class label of t. CL is used to keep track of the tuples location (i.e., in which leaf nodes) as leaf nodes are split.

Class List Extension. In order to support data classification with free tuples where their class labels are assigned dynamically, we need to extend SLIQ with the following modifications. The class list table $CL(nid, cid, sid)$ is extended with an additional column "sid", which represents a subset identifier, to indicate which subset (i.e., P_i) a tuple belongs to. This additional information is needed to determine the optimal Gini index values as discussed in the previous section. Consider a tuple t that is the i^{th} tuple in D, the cid and sid values in CL are maintained as follows. If t belongs to J_0, then $CL[i].cid = 0$ and $CL[i].sid = 0$. If t is a bound tuple in P_j, then $CL[i].cid = 1$ and $CL[i].sid = j$. Otherwise, if t is a free tuple in P_j, then $CL[i].cid = -1$ and $CL[i].sid = j$.

Example 3.5 *Figure 3-3 shows some data structures created for computing IEQs for $Q_4(D)$. Figure 3-3(a) shows the attribute list created for attribute Master.name; and Figure 3-3(d) shows the initial class list created for J_{hub}, where all the records are in a single leaf node (with nid value of 1).* □

Naive TALOS (TALOS$^-$). Before presenting the optimizations for TALOS in the next section, let us first describe a non-optimized, naive variant of TALOS (denoted by TALOS$^-$). Suppose that we are considering an IEQ Q' where $rel(Q') = \{R_1, \cdots, R_n\}$, $n > 1$, that is derived from some schema subgraph G. First, TALOS$^-$ joins all the relations in $rel(Q')$ (based on the foreign-key joins represented in G) to obtain a single relation J. Next, TALOS$^-$ computes attribute lists for the attributes in J and a class list for J. TALOS$^-$ is now ready to construct a decision tree DT to derive the IEQ Q' with these structures. The decision tree DT is initialized with a single leaf node consisting of the records in J, which is then refined iteratively by splitting the leaf nodes in DT. TALOS$^-$ terminates the splitting of a leaf node when (1) its tuples are either all labeled positive or all labeled negative; or (2) its tuples have the same attribute values w.r.t. all the splitting attributes. Finally, TALOS$^-$ classifies each leaf node in DT as positive or negative as follows: a leaf node is classified as positive if and only if the ratio of the number of its negative tuples to the number of its positive tuples is smaller than a threshold value given by τ [4]. The selection condition of the IEQ Q' is then derived from the collection of positive leaf nodes in DT as follows. Each internal node in DT corresponds to a selection predicate on some attribute of J, and each root-to-positive-leaf path P_j in DT corresponds to a conjunctive predicate C_j on J. Thus, each decision tree enumerated for G yields a selection predicate for Q' of the form C_1 *or* C_2 $\cdots$ *or* C_ℓ. In the event that all the leaf nodes in DT are classified as negative, the computation of Q' is not successful (i.e., there is no IEQ for $rel(Q')$), and we refer to Q' as a *pruned IEQ*.

3.3.2 Optimizations

The naive TALOS described in the previous section suffers from two drawbacks. First, the overhead of computing J can be high; especially if there are many large relations in $rel(Q')$. Second, since the cardinality of J can be much larger than the cardinality of each of the relations in $rel(Q')$, building decision trees directly using J entails the computation and scanning of correspondingly large attribute lists, which further increases the computation cost. In the rest of this section, we present the optimization techniques used by TALOS to address the above performance issues.

Join Indices & Hub Table. To avoid the overhead of computing J from $rel(Q')$, TALOS exploits pre-computed join indices [52], which is a well-known technique for optimizing joins. For each pair of relations, R and R', in the database schema that are related by a foreign-key join, its join index, denoted by $I_{R,R'}$, is a set of pairs of row identifiers referring to a record in each of R and R' that are related by the foreign-key join.

Based on the foreign-key join relationships represented in the schema subgraph G, TALOS computes the join of all the appropriate join indices for $rel(Q')$ to derive a relation, called the *hub table*, denoted by J_{hub}. Computing J_{hub} is much more efficient than computing J, since there are fewer number of join operations (i.e., number of relevant join indices) and each join attribute is a single integer-valued column.

[4]In our experiments, we set $\tau = 1$.

Example 3.6 *Consider again query Q_4 introduced in Example 3.2. Suppose that we are computing IEQ Q'_4 with rel(Q'_4) = {Master, Batting, Team}. Figure 3-3(b) shows the hub table, J_{hub}, produced by joining two join indices: one for Master $\bowtie_{pID}$ Batting, and the other for Batting $\bowtie_{team,year}$ Team. Here, r_M, r_B, and r_T refer to the row identifiers for Master, Batting, and Team relations, respectively.* □

Mapping Tables. Instead of computing and operating on large attribute lists (each with cardinality equal to $|J|$) as in the naive approach, TALOS operates over the smaller pre-computed attribute lists AL_i for the base relations in $rel(Q')$ together with small *mapping tables* to link the pre-computed attribute lists to the hub table. In this way, TALOS only needs to pre-compute once the attribute lists for all the base relations, thereby avoiding the overhead of computing many large attribute lists for different $rel(Q')$ considered.

Each mapping table, denoted by M_i, is created for each $R_i \in rel(Q')$ that links each record r in R_i to the set of records in J_{hub} that are related to r. Specifically, for each record r in R_i, there is one record in M_i of the form (j, S), where j is the row identifier of r, and S is a set of row identifiers representing the set of records in J_{hub} that are created from r.

Example 3.7 *Figure 3-3(c) shows the mapping table M_{Master} that links the Master relation in Figure 3-1 and J_{hub} in Figure 3-3(b). The record $(2, \{2, 3\})$ in M_{Master} indicates that the second tuple in Master relation (with pID of P2), contributed to two tuples, located in the second and third rows, in J_{hub}.* □

Computing Class List. We now explain how TALOS can efficiently compute the class list CL for J (without having explicitly computed J) by using the attribute lists, hub table, and mapping tables. The key task in computing CL is to partition the records in J into subsets (J_0, P_1, P_2, etc.), as described in the previous section.

For simplicity and without loss of generality, assume that the schema of $Q(D)$ has n attributes $A_1, \cdots, A_n$, where each A_i is an attribute of relation R_i. TALOS first initializes CL with one entry for each record in J_{hub} with the following default values: $nid = 1$,

$cid = 0$, and $sid = 0$. For each record r_k that is accessed by a sequential scan of $Q(D)$, TALOS examines the value v_i of each attribute A_i of r_k. For each v_i, TALOS first retrieves the set of row identifiers RI_{v_i} of records in R_i that have a value of v_i for attribute $R_i.A_i$ by performing a binary search on the attribute list for $R_i.A_i$. With this set of row identifiers RI_{v_i}, TALOS probes the mapping table M_i to retrieve the set of row identifiers JI_{v_i} of the records in J_{hub} that are related to the records referenced by RI_{v_i}. The intersection of the JI_{v_i}'s for all the attribute values of r_k, denoted by P_k, represents the set of records in J that can generate r_k. TALOS updates the entries in CL corresponding to the row identifiers in P_k as follows: (1) the *sid* value of each entry is set to k (i.e., all the entries belong to the same subset corresponding to record r_k), and (2) the *cid* value of each entry is set to 1 (i.e., tuple is labeled positive) if $|P_k| = 1$; otherwise, it is set to -1 (i.e., it is a free tuple).

Example 3.8 *We illustrate how* TALOS *creates CL for query Q_4, which is shown in Figure 3-3(d). Initially, each row in CL is initialized with sid $= 0$ and cid $= 0$.* TALOS *then accesses each record of $Q_4(D)$ sequentially. For the first record (with name = "B'),* TALOS *searches AL_{name} and obtains $RI_B = \{2\}$. It then probes M_{Master} with the row identifier in RI_B, and obtains $JI_B = \{2, 3\}$. Since $Q_4(D)$ contains only one attribute, we have $P_1 = \{2, 3\}$. The second and the third rows in CL are then updated with sid $= 1$ and cid $= -1$. Similarly, for the second record in $Q_4(D)$ (with name = "E"),* TALOS *searches AL_{name} and obtains $RI_E = \{5\}$, and derives $JI_E = \{6\}$ and $P_2 = \{6\}$. The sixth row in CL is then updated with sid $= 2$ and cid $= 1$.* □

3.4 Ranking IEQs

In this section, we describe the ranking criteria we adopt to prioritize results presented to the user. Specifically, we consider a metric based on the Minimum Description Length (MDL) principle [41], and two metrics based on the F-measure [53].

3.4.1 Minimum Description Length

The Minimum Description Length (MDL) principle argues that all else being equal, the best model is the one that minimizes the sum of the cost of describing the data given the model and the cost of describing the model itself. If M is a model that encodes the data D, then the total cost of the encoding, $cost(M, D)$, is defined as: $cost(M, D) = cost(D|M) + cost(M)$. Here, $cost(M)$ is the cost to encode the model (i.e., the decision tree in our case), and $cost(D|M)$ is the cost to encode the data given the model. We can rely on succinct tree-based representations to compute $cost(M)$. The data encoding cost, $cost(D|M)$, is calculated as the sum of classification errors. The details of the encoding computations are given elsewhere [32]. The smaller the MDL of an IEQ is, the better the query is.

3.4.2 F-measure

We now present two useful metrics based on the popular F-measure [53] that represents the precision of the IEQs. The first variant follows the standard definition of F-measure: the F-measure for two IEQs Q and Q' is defined as $F_m = \frac{2\times|p_a|}{2\times|p_a|+|p_b|+|p_c|}$, where $p_a = Q(D) \cap Q'(D)$, $p_b = Q'(D) - Q(D)$, and $p_c = Q(D) - Q'(D)$. We denote this variant as *F-measure* in our experimental study. In contrast to the MDL metrics, the higher the F-measure of an IEQ is, the more precise the query is and therefore the better the query is.

Observe that the first variant of F-measure is useful only for approximate IEQs, and is not able to distinguish among precise IEQs, as this metric gives identical values for precise IEQs since p_b and p_c are empty. To rank precise IEQs, we introduce a second variant, denoted by F_m^{est}, which relies on estimating p_a, p_b, and p_c using some data probabilistic models (as opposed to using the actual values from the data set). F_m^{est} captures how the equivalence of queries is affected by database updates, and the IEQ with high F_m^{est} is preferable to another IEQ with low F_m^{est}. For simplicity, we use a simple independent model to estimate F_m^{est}; other techniques such as the Bayesian model by Getoor and others [16] can be applied too. The second variant has the benefit that estimates, which are computed from a global distribution model, may more accurately reflect the true relevance of the IEQs than one computed directly from the data. This of course pre-supposes that future updates follow the existing data distribution.

Details on computing the F-measure (F_m and F_m^{est}) are given in Section 3.5.

3.5 Implementation of TALOS

In this section, we describe the implementation details of TALOS and analyze the running time and the space complexity of TALOS. We also discuss some control knobs that can be used to restrict the search space in TALOS.

3.5.1 Implementation

TALOS is implemented at the application level and interacts with the DBMS by issuing appropriate SQL queries. The architecture of TALOS is depicted in Figure 3-4, and its procedures are sketched in Algorithm 1. We now elaborate on the details of the steps of TALOS to derive the IEQs of a given input query Q and a database D.

TALOS first computes the query result $Q(D)$ by posing Q into the DBMS (line 1). We choose this simple implementation to ease TALOS from evaluating the input query Q, which can be in complex fragments (e.g., SQL query with sub-queries). The next step of TALOS is to enumerate different schema subgraphs G, each of which contains a set of core relations $\mathcal{R}$ of Q. For each derived schema subgraph G, TALOS computes the following four main data structures, including (S1) to (S4).

(S1) The hub table (J_{hub}), which is stored as a view in the DBMS. TALOS computes J_{hub} by issuing a query Q_{hub} that joins the join indices corresponding to the edges in the corresponding schema subgraph G that is being considered (line 6).

(S2) The mapping table (M_i) between each relation $R_i \in G$ and J_{hub}. Each M_i is derived by issuing a query Q_{map} to scan the column of J_{hub} corresponding to R_i (line 7).

(S3) The attribute list (AL_A) for each splitting attribute that can be used in $sel(Q')$. By default, TALOS uses all attributes in each relation $R_i \in G$ except the primary keys as potential splitting attributes for the IEQs to be interesting[5]. Each attribute list AL_A of an attribute $A \in R_i$ is derived by issuing a query Q_{al} to retrieve tuples from the column A of the corresponding relation R_i (line 9). Note that TALOS has a knob to control the set of attributes that can be used in $sel(Q')$ (to be discussed later).

(S4) The class list (CL) is derived using the attribute lists of attributes in $proj(Q)$ and $Q(D)$ (line 10).

Among these structures, the hub tables and attribute lists are stored inside the DBMS; the class list and mapping tables are stored in the main memory. If the main memory is large enough (we present how to compute the memory usage of TALOS in Section 3.5.2), TALOS will store the attribute lists and/or hub tables in the main memory to enhance the performance. After these data structures have been built, TALOS proceeds to build the decision trees DT to derive the IEQs w.r.t. each schema subgraph G (line 11). The basic task is to scan all attribute lists at each leaf node of the decision tree to determine the optimal node split[6]. After deriving a decision tree DT from G, if users want to find more than one decision tree w.r.t. G, TALOS removes the splitting attribute that is used in the first level of DT[7] from the set of possible splitting attributes, and derives other DT's from G. The default setting of TALOS is to try computing more than one IEQs from a schema subgraph G; TALOS allows users to disable this option if necessary. Lastly, TALOS derives the IEQ Q' from each computed DT, and calculates its MDL and F-measure metrics (line 13 - 14).

To complete the discussion of TALOS's procedures, we will explain how TALOS enumerates schema subgraphs containing a set of core relations of Q (line 4), and computes the F-measure of a derived IEQ (line 14) next.

[5]It is trivial to require TALOS to use the primary keys in $sel(Q')$ also.

[6]TALOS actually scans each attribute list one time to compute the optimal node splits at all the leaf nodes that are being considered simultaneously to improve the performance.

[7]Another possibility is to remove all the splitting attributes used in DT.

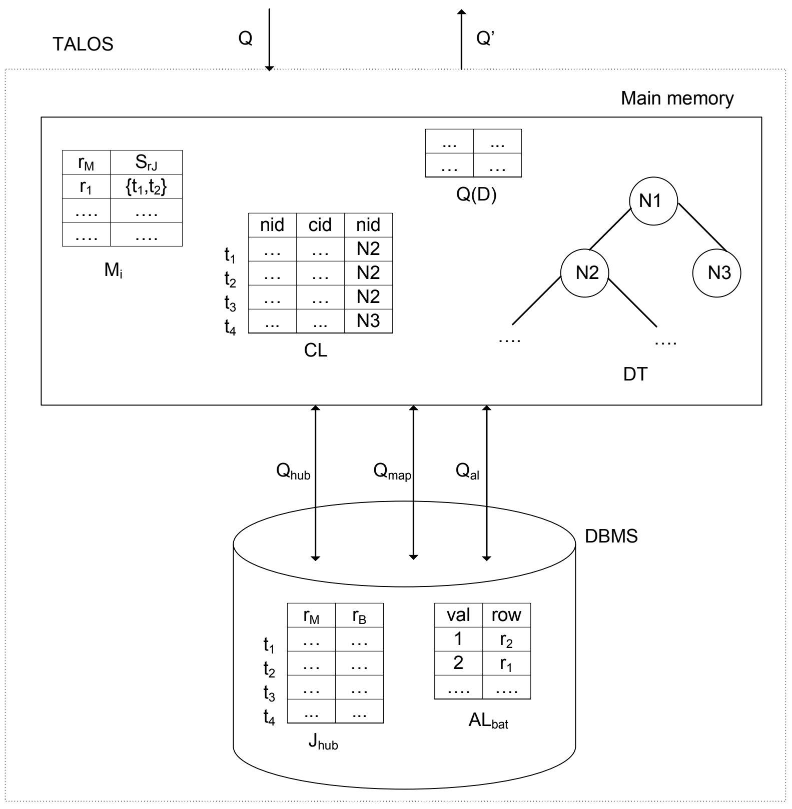


Figure 3-4: The Architecture of TALOS

Algorithm 1: TALOS(Q, D)

```
1  Compute the query result $Q(D)$;
2  Compute sets of core relations of $Q$;
3  foreach set of core relations $\mathcal{R}$ do
4      Enumerate schema subgraphs $G$ containing $\mathcal{R}$;
5      foreach schema subgraph $G$ do
6          Compute $J_{hub}$ corresponding to $G$;
7          Derive mapping tables $M_i$ between each relation $R_i \in G$ and $J_{hub}$;
8          Enumerate splitting attributes that can be used in $sel(Q')$;
9          Compute attribute lists $AL_A$ for each splitting attribute $A$;
10         Compute the class list $CL$ for $J_{hub}$;
11         Build the decision trees on $J_{hub}$;
12         foreach derived decision tree $DT$ do
13             Derive the corresponding IEQ $Q'$;
14             Compute the MDL and F-measure of $Q'$;
15         end
16     end
17 end
```

Enumerating Schema Subgraphs. The technique of TALOS to enumerate schema subgraphs G containing a set of core relation ($\mathcal{R}$) is based on a bread-first search traversal of the schema graph $\mathcal{SG}$ starting from an arbitrary vertex R_s in $\mathcal{R}$. TALOS keeps a queue $\mathcal{QG}$ of "active" schema subgraphs; $\mathcal{QG}$ is initialized with one schema subgraph G_s containing the vertex R_s. In each round, TALOS picks from $\mathcal{QG}$ an active schema subgraph G, and outputs G as a derived schema subgraph if G contains vertices corresponding to all the core relations in $\mathcal{R}$. TALOS also expands G into larger subgraphs G' by adding one edge that connects a vertex V in G with a neighbor of V that is currently not in G, and places the resultant graph G' into $\mathcal{QG}$. TALOS introduces a control knob, denoted as n_{max}, to constraint the maximum number of vertices in a derived subgraph not to exceed n_{max} for efficiency reason. Therefore, TALOS will not expand a subgraph G if the number of vertices in G is equal or greater than n_{max}.

Computing F-measure. To compute the F-measure of an IEQ Q' derived from a decision tree DT, we observe that the set of tuples selected by Q', denoted as $J_{Q'}$, has been collected in the corresponding class list CL. Thus, TALOS scans the class list CL to derive $J_{Q'}$, where a tuple $t \in CL$ belongs to $J_{Q'}$ if $t.nid$ is one of the identifiers of the leaf nodes in DT that are used to derive Q'.

With the presence of $J_{Q'}$, TALOS scans the attribute lists of attributes in $proj(Q)$ to derive $Q'(D)$ as follows. For each row $r_k = (v, r)$ in the attribute list AL_A with $A \in proj(Q)$ and M_i is the mapping table between the relation R_i containing A and J_{hub}, TALOS probes M_i with the value r to retrieve the set of row identifiers JI_v of the records in J_{hub} that are related to r_k. For every row identifier $rid \in JI_v$ such that the corresponding tuple t_r of rid belongs to $J_{Q'}$, TALOS updates $t_r.A_i = v$. After $Q'(D)$ has been derived, TALOS computes the F-measure of Q' using its definition given in Section 3.4.2.

Computing F_m^{est}. We present how TALOS computes the F_m^{est} for a pair of IEQs Q and Q'. For simplicity and without loss of generality, assume that $sel(Q) = C_1 \ or \ \cdots \ or \ C_\ell$ and $sel(Q') = C'_1 \ or \ \cdots \ or \ C'_m$.

To compute the F_m^{est} of Q', TALOS estimates $|p_a|$ as the probability in the event that an inserted tuple τ into the database satisfies both Q and Q'. Thus, TALOS estimates $|p_a|$ as the probability that τ satisfies at least one predicate C_i in $sel(Q)$ and at least one predicate C'_j in $sel(Q')$. In other words, $|p_a|$ is computed: $|p_a| = \sum_{C_i \in sel(Q)}^{C_j \in sel(Q')} s(C_i \wedge C_j)$, where $s(P_i)$ denotes the selectivity of the selection condition P_i. TALOS uses a simple independent model to estimate the selectivity of $(C_i \wedge C_j)$, which is the product of the selectivity of C_i and C_j.

TALOS estimates $|p_b|$ and $|p_c|$ in the same way as what have explained for computing $|p_a|$. For instance, since p_b represents the set of tuples that satisfy Q and do not satisfy Q', $|p_b|$ is estimated as the probability for the event that an inserted tuple τ satisfies $sel(Q)$ but does not satisfy $sel(Q')$. In other words, $|p_b|$ is the probability that τ satisfies both $sel(Q)$ and $\neg sel(Q')$.

3.5.2 Complexity Analysis

We now analyze the running time and the space complexity (in the worst case) of TALOS to derive IEQs for a given input query Q w.r.t. a schema subgraph G containing n relations $R_1, \cdots, R_n$. We further assume that the set of projected attributes of Q consists of k attributes $A_1, \cdots, A_k$, where each A_i is an attribute of relation R_i. In the following, we use $|X|$ to denote the number of tuples in a relation X.

Time Complexity. The running time of TALOS is proportional to the summation of the following four main components, including (T1) to (T4).

(T1) The time T_{hub}^{talos} to compute the hub table J_{hub} corresponding to G. T_{hub}^{talos} is computed depending on the join algorithms used inside the DBMS.

(T2) The time T_{al}^{talos} to derive attribute lists. T_{al}^{talos} is in the order of $\sum_{i=1}^{n} n_i(|R_i| \log |R_i|)$ with n_i denotes the number of attributes in R_i, since TALOS needs to sort order all the attribute lists for the task of computing optimal node splits.

(T3) The time T_{cl}^{talos} to derive the class list CL. Recall that the basic step of TALOS to derive CL is for each row $r = (v_1, \cdots, v_k)$ of $Q(D)$, TALOS probes the corresponding attribute list of A_i with a value v_i in $O(\log |R_i|)$ time, and intersects the sets of derived row identifiers of J_{hub} corresponding each v_i's. The intersection operation runs in $O(|J_{hub}|)$ in the worst case and $O(1)$ in the best case. Hence, T^{talos}_{cl} is in the order of $O(|Q(D)| \sum_{i=1}^{k} \log |R_i| + |Q(D)||J_{hub}|)$.

(T4) The time T^{talos}_{dt} to build a single decision tree. If `TALOS` derives more than one decision tree, then T^{talos}_{dt} increases in proportion with the number of the decision trees derived. Since `TALOS` needs to scan all the attribute lists to derive the optimal splitting condition at each leaf node that is being considered, T^{talos}_{dt} is in the order of $O(\ell \sum_{i=1}^{n} n_i |R_i| \mathtt{C_{gini}})$, where ℓ is the maximum height of the derived decision tree, and $\mathtt{C_{gini}}$ is the computation cost to derive the optimal Gini value. We have $\mathtt{C_{gini}} = O(|Q(D)|)$, since `TALOS` basically updates the number of free tuples in each partition of the class list CL that have been split into the left and right child nodes to derive the domain of f_1, f_2[8] for the computation of the optimal Gini value.

Space Complexity. Similar to `SLIQ`, `TALOS` keeps the class list CL in the main memory and the attribute lists in the disk when there is not enough available main memory. The mapping tables are kept in the main memory for `TALOS` to look up whenever scanning the attribute lists.

The space complexity of `TALOS` can be computed as follows. Since the mapping table between a relation R_i and J_{hub} contains $(|J_{hub}| + |R_i|)$ integer values, the total size for the mapping tables is proportional to $(n|J_{hub}| + \sum_{i=1}^{n} |R_i|)$. The size of the class list is also proportional to $|J_{hub}|$. Thus, the space complexity of `TALOS` is in the order of $((n + 1)|J_{hub}| + \sum_{i=1}^{n} |R_i|)$.

In the event that the available main memory cannot account for all the mapping tables, `TALOS` selects the mapping tables in the decreasing order of their memory's usage to force these mapping tables into the disk until the memory is enough to store the remaining data structures. Without loss of generality, assume that the mapping tables of

[8]Recall that f_1 (resp. f_2) represents the number of free tuples to be labeled positive in the left (resp. right) child node.

relations $R_1, \cdots, R_m$ need to be disk-resident. In this case, TALOS updates the attribute list AL_A of every attribute $A \in R_i$, $i \in [1, m]$, by joining AL_A with the mapping table M_i so that the resultant AL_A contains tuples in the form (v, t), where t refers to the row identifiers of tuples in J_{hub} instead of row identifiers in R_i. The purpose of this step is to avoid probing each row of AL_A, for $A \in R_i$ and $i \in [1, m]$, with the mapping tables in the disk.

3.5.3 Control Knobs

As the search space for IEQs can be very large, particularly with large complex database schema where each relation has foreign-key joins with other relations, users should be able to restrict the search space by specifying hints/preferences in the form of control parameters. Some examples include the following four control knobs.

(K1) Constraining the number of relations in the from-clause of each IEQ to be in the range $[n_{min}, n_{max}]$. This control knob constraints the number of vertices in each derived schema subgraph to be in the range $[n_{min}, n_{max}]$.

(K2) Constraining the number of selection predicates in each conjunction of $sel(Q')$ in the range $[h_{min}, h_{max}]$. Recall that the selection condition of an IEQ is presented in the disjunctive normal form "C_1 or $C_2 \cdots$ or C_ℓ", where each C_i is a conjunction of selection predicates. Thus, this control knob constraints the number of predicates in each C_i to be in the range $[h_{min}, h_{max}]$. This knob is implemented by setting the height of the derived decision trees to be in the range $[h_{min}, h_{max}]$.

(K3) Specifying a specific set of relations to be included (excluded) in (from) Q'. This knob is imposed in the step of enumerating different schema subgraphs containing core relations.

(K4) Specifying a specific set of attributes to be included (excluded) in (from) the selection predicates in Q'. This knob is implemented by not deriving the attribute lists for attributes that are required to be excluded from $sel(Q')$, and forcing the decision tree to use the attributes to be included in $sel(Q')$ for the splitting conditions.

3.6 Experimental Study

In this section, we evaluate the performance of our proposed approaches for computing IEQs and study the relevance of the results returned. The algorithms being compared include our proposed `TALOS` approach, which is based on a dynamic assignment of class labels for free tuples, and two static class labeling techniques: `NI` labels all the free tuples as positive, and `RD` labels a random number of at least one free tuple in each subset as positive. We also examine the effectiveness of our proposed optimizations by comparing against a non-optimized naive variant of `TALOS` (denoted by `TALOS`$^-$) described in Section 3.3.1.

The database system used for the experiments is MySQL Server 5.0.51; and all algorithms are coded using C++ and compiled and optimized with GNU C++ compiler. Our experiments are conducted on dual core 2.33GHz machine with 3.25GB RAM and a 250GB hard disk, running Linux. The experimental result timings reported are averaged over 5 runs with caching effects removed.

3.6.1 At-Least-One Semantic Metric

To represent the *flexibility* in dynamically assigning class labels for free tuples under the *at-least-one* semantics, we use the following metric, called the **a**verage number of **f**ree tuples per one **p**artition (`afp`). The `afp` of an input query Q is computed as the ratio between the total number of free tuples and the number of tuples in $Q(D)$. More precisely, given a query Q, let J denote the join result of joining all relations in $rel(Q)$ using their foreign-key joins. Let $J_1 \subseteq J$ be the maximal set of tuples corresponding to $Q(D)$; i.e., $\pi_{proj(Q)}(J_1) = Q(D)$. The `afp` of Q is computed as $\frac{|J_1|}{|Q(D)|}$. Intuitively, `afp` represents the number of free tuples, of which at least one tuple must be labeled positive to produce the corresponding tuple in the query result of the IEQs. The higher the `afp` of a query Q is, the more flexibility we have to derive the IEQs for Q.

Table	# Tuples
adult	45222
Master	16639
Batting	88686
Pitching	37598
Fielding	128426
Salaries	18115
Team	2535
Manager	3099

(a) Adult & Baseball

Table	Symbol	# Tuples
lineitem	L	6001215
order	O	1500000
partsupp	PS	800000
part	P	200000
customer	C	150000
supplier	S	10000
nation	N	25

(b) TPCH

Table 3.3: Table sizes (number of tuples)

	Query
A_1	$\pi_{nc}\ \sigma_{occ=\text{“Armed-Force”}}\ (adult)$
A_2	$\pi_{edu,occ}\ (\sigma_{ms=\text{“Never-married”} \wedge 64 \leq age \leq 68 \wedge race=\text{“White”} \wedge gain>500 \wedge sex=\text{“F”}}\ adult)$
A_3	$\pi_{nc,gain}\ (\sigma_{ms=\text{“Never married”}}\ adult)$
A_4	$\pi_{id}\ (\sigma_{SKY\text{-}LINE(gain\ MAX, age\ MIN)}\ adult)$
B_1	$\pi_{M.name}\ (\sigma_{team=\text{“ARI”} \wedge year=2006 \wedge HR>10}\ (Master \bowtie Batting))$
B_2	$\pi_{M.name}\ (\sigma_{sum(HR)>600}\ (Master \bowtie Batting))$
B_3	$\pi_{M.name}\ (\sigma_{SKY\text{-}LINE(HR\ MAX, SO\ MIN)}\ (Master \bowtie Batting))$
B_4	$\pi_{M.name,T.year,T.rank}\ (\sigma_{team=\text{“CIN”} \wedge 1982<year<1988}\ (Manager \bowtie Team))$
T_1	$\pi_{S.name,N.name}\ \sigma_{S.acctbal>4000 \wedge N.regionkey<4}\ (supplier \bowtie nation)$
T_2	$\pi_{C.name,N.name}\ \sigma_{C.acctbal>3000}\ (customer \bowtie nation)$
T_3	$\pi_{P.name,S.name}\ \sigma_{PS.avaiqty>3000 \wedge S.acctbal>9500}\ (part \bowtie partsupp \bowtie supplier)$
T_4	$\pi_{O.clerk,L.extendedprice}\ \sigma_{L.quantity<2 \wedge O.orderstatus=\text{“P”}}\ (lineitem \bowtie order)$

Table 3.4: Test queries for experiments with TALOS

3.6.2 Data sets and Queries

We use three real data sets: one small size (Adult), one medium size (Baseball), and one large data set (TPCH). The size of the test data is shown in Table 3.3, and the test queries are given in Table 3.4. The characteristics of test queries are shown in Table 3.5, where columns 2 & 3 indicate the number of tuples in the query result of Q and its core query Q_S[9], respectively. The last column shows the average number of free tuples per one partition (`afp`) of Q.

Adult. The Adult data set, from the UCI Machine Learning Repository[10], is a single-

[9]Recall that the core query Q_S of Q is derived from Q by replacing $proj(Q)$ by a set S of primary keys of core relations containing attributes in $proj(Q)$.

[10]http://archive.ics.uci.edu/ml/datasets/Adult

Query	$\lvert Q(D)\rvert$	$\lvert Q_S(D)\rvert$	afp
A_1	1	14	41292
A_2	4	5	637
A_3	137	15000	312
A_4	4	4	1
B_1	7	7	8.3
B_2	4	4	22.3
B_3	35	35	8.8
B_4	6	6	1
T_1	4383	4383	1
T_2	95264	95264	1
T_3	24672	24672	1
T_4	3719	3719	1

Table 3.5: The characteristics of test queries

relation data set that has been used in many classification works. We use this data set to illustrate the utility of the IEQs for the simple case when both the input query Q as well as the output IEQ Q' involve only one relation. The four test queries for this data set are A_1, A_2, A_3 and A_4[11]. The first three queries have different `afp`'s values: very high (A_1), high (A_2), and medium (A_3). Query A_4 is used to illustrate how `TALOS` handles skyline queries.

In addition, we also run three sets of workload queries with varying average number of free tuples per one partition (`afp`) factor (very high, high, medium) as shown in Table 3.6. Each workload set W_i consists of five queries denoted by W_{i1} to W_{i5}. The average `afp` of the queries in W_1, W_2, and W_3 are, respectively, 14243, 325, and 60.

Baseball. The baseball data set is a more complex, multi-relation database that contains Batting, Pitching, and Fielding statistics for Major League Baseball from 1871 through 2006 created by Sean Lahman. The queries used for this data set (B_1, B_2, B_3, B_4) are common queries that mainly relate to baseball players' performance. The `afp` of these queries is low and in the order of 10.

TPC-H. To evaluate the scalability of our approach, we use the TPC-H data set (with a

[11] We use *gain*, *ms*, *edu*, *loss*, *nc*, *hpw*, and *rs*, respectively, as abbreviations for capital-gain, marital-status, education, capital-loss, native-country, hours-per-week, and relationship.

	Query	`afp`
W_{11}	π_{ms} ($\sigma_{19 \leq age \leq 22 \wedge edu=\text{“Bachelors”}}$ $adult$)	17826
W_{12}	π_{nc} ($\sigma_{occ=\text{“Armed-Force”}}$ $adult$)	41292
W_{13}	$\pi_{occ,ms}$ ($\sigma_{nc=\text{“Phillipines”} \wedge 30 \leq age \leq 40}$ $adult$)	1106
W_{14}	π_{edu} ($\sigma_{ms=\text{“Married-AF”} \wedge race=\text{“Asian”}}$ $adult$)	7570
W_{15}	π_{edu} ($\sigma_{23 \leq age \leq 24 \wedge nc=\text{“Germany”}}$ $adult$)	3422
W_{21}	$\pi_{occ,edu}$ ($\sigma_{gain>9999}$ $adult$)	447
W_{32}	$\pi_{occ,edu}$ ($\sigma_{ms=\text{“NM”} \wedge 64 \leq age \leq 68 \wedge race=\text{“White”}}$ $\sigma_{gain>500 \wedge sex=\text{“F”}}$ $adult$)	637
W_{23}	$\pi_{age,wc,edu}$ ($\sigma_{hpw \leq 19 \wedge race=\text{“White”} \wedge nc=\text{“England”}}$ $adult$)	197
W_{24}	$\pi_{edu,age}$ ($\sigma_{ms=\text{“Separated”} \wedge wc=\text{“State-gov”}}$ $\sigma_{race=\text{“White”}}$ $adult$)	238
W_{25}	$\pi_{edu,age}$ ($\sigma_{wc=\text{“Private”} \wedge race=\text{“Asian”}}$ $adult$)	107
W_{31}	π_{age} ($\sigma_{ms=\text{“Divorced”} \wedge wc=\text{“State”} \wedge age>70}$ $adult$)	82
W_{32}	$\pi_{age,wc,edu}$ ($\sigma_{hpw \leq 19 \wedge race=\text{“White”}}$ $adult$)	42
W_{33}	$\pi_{edu,age,ms}$ ($\sigma_{wc=\text{“Private”} \wedge race=\text{“Asian”}}$ $adult$)	54
W_{34}	$\pi_{edu,age,gain}$ ($\sigma_{ms=\text{“Married-civ”} \wedge race\text{“Asian”}}$ $\sigma_{30 \leq age \leq 37}$ $adult$)	109
W_{35}	$\pi_{edu,gain}$ ($\sigma_{gain>5000 \wedge nc=\text{“Vietnam”}}$ $adult$)	11

Table 3.6: Workload query sets for Adult

size of 1GB), and four test queries T_1 - T_4, involving large relations.

Control Knobs. In our experiments, we set the following three control knobs, as shown in Table 3.7, for efficiency reason : (K1) the number of relations in the from-clause of the IEQs, (K2) the number of selection predicates in each conjunction of $sel(Q')$, and (K4) the attributes used in $sel(Q')$. For Adult and Baseball data set, we use the default setting that allows any attributes (except the primary keys) to appear in the selection predicates of the IEQs. For TPCH, the attributes that can be used in the selection predicates of the IEQs are selected from the set S_{tpch} = {C.acctbal, C.mktsegment, O.orderstatus, O.orderpriority, PS.availqty, PS.supplycost, S.acctbal, P.retailprice, N.regionkey, L.quantity}.

Control knob	Adult	Baseball	TPCH
(K1) # relations in the from-clause	$[1, 1]$	$[2, 4]$	$[2, 3]$
(K2) # selection predicates in each conjunction of $sel(Q')$	$[0, \infty)$	$[0, \infty)$	$[0, 4]$
(K4) Attributes in $sel(Q')$	all	all	S_{tpch}

Table 3.7: The control knob values

3.6.3 Comparing TALOS, NI, and RD

In this section, we compare `TALOS` against the two static class labeling schemes, `NI` and `RD`, in terms of their efficiency as well as the quality of the generated IEQs.

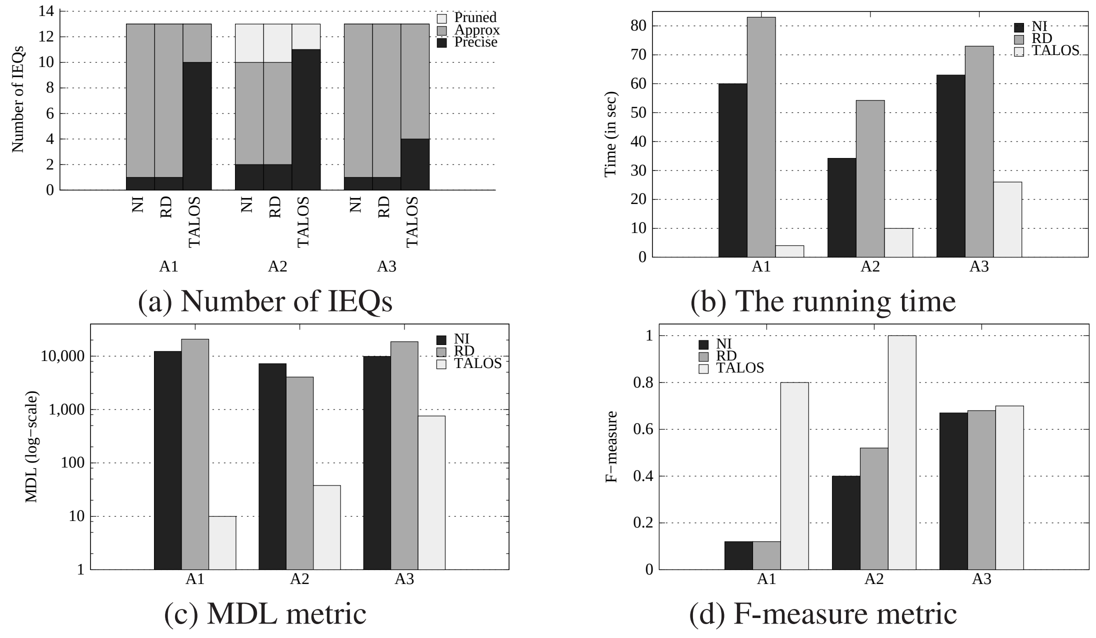
Figures 3-5(a) and (b) compare the performance of the three algorithms in terms of the number of weak IEQs generated and their running times, respectively, using the queries A_1 to A_4. Note that Figure 3-5 only compares the performance for weak IEQs because the Adult data set is a single-relation database, all the tuples are necessarily bound when computing strong IEQs. Thus, the performance results for strong IEQs are the same for all algorithms and are therefore omitted. Similarly, the results for query A_4 are also omitted from the graphs because it happens that all the tuples are bound for query A_4; hence, the performance results are again the same for all three algorithms.

The results in Figures 3-5(a) and (b) clearly show that `TALOS` outperforms `NI` and `RD` in terms of both the total number of (precise and approximate) IEQs computed[12] as well as the running time. In particular, observe that the number of precise IEQs from `TALOS` is consistently larger than that from `NI` and `RD`. The *flexibility* of dynamic assignment of class labels for free tuples increases `TALOS`'s opportunities to derive precise IEQs. In contrast, the static class label assignment schemes of `NI` and `RD` are too restrictive and are not effective for generating precise IEQs.

In addition, `TALOS` is also more efficient than `NI` and `RD` in terms of the running time. The reason for this is due to the flexibility of `TALOS`'s dynamic labeling scheme for free tuples, which results in decision trees that are smaller than those constructed by `NI` and `RD`. Table 3.8 compares the decision trees constructed by `TALOS`, `NI`, and `RD` in terms of their average height and average size (i.e., number of nodes). Observe that the decision trees constructed by `TALOS` are significantly more compact than that by `NI` and `RD`.

Figures 3-5(c) and (d) compare the quality of the IEQs generated by the three algo-

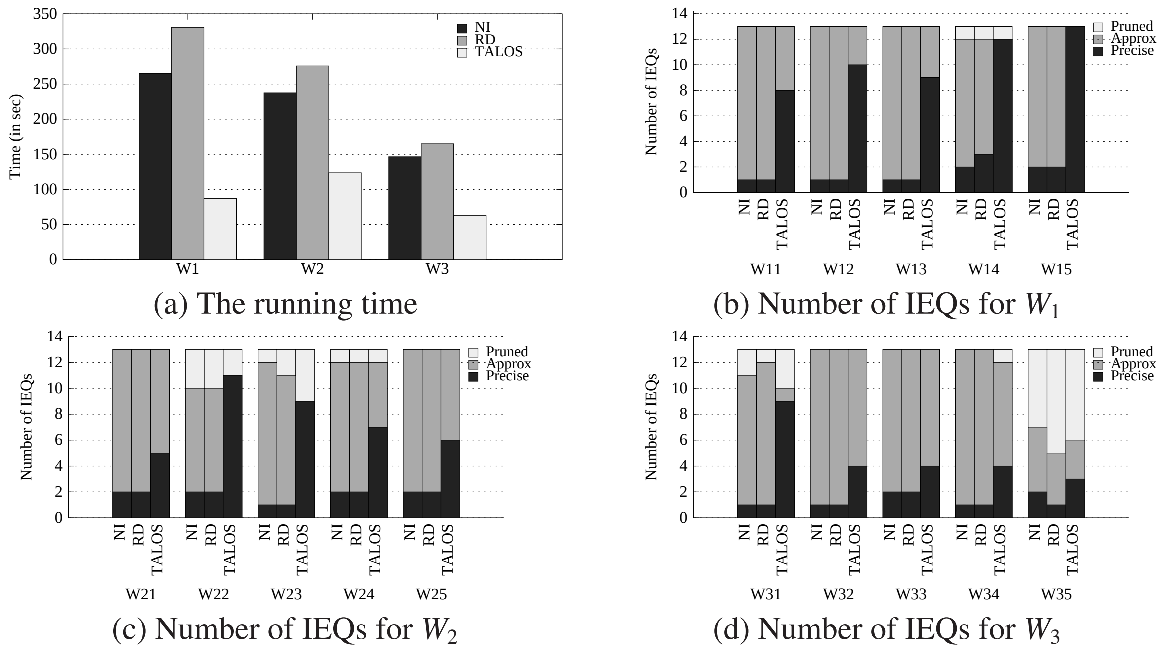
[12]For clarity, we have also indicated in Figure 3-5(a) the number of pruned IEQs (defined in Section 3.3.1) computed by each algorithm. Since the number of decision trees considered by all three algorithms is the same, the sum of the number of precise, approximate, and pruned IEQs generated by all the algorithms are the same.



(a) Number of IEQs (b) The running time

(c) MDL metric (d) F-measure metric

Figure 3-5: Comparison of TALOS, NI and RD for queries in Adult



(a) The running time (b) Number of IEQs for W_1

(c) Number of IEQs for W_2 (d) Number of IEQs for W_3

Figure 3-6: Comparison of TALOS, NI and RD for workload queries

Query	Average height			Average size		
	NI	RD	TALOS	NI	RD	TALOS
A_1	14.9	19.8	2.1	5304	9360	4.7
A_2	16.1	21.8	6.5	3224	2970	19.2
A_3	16	20	12	4386	8103	334

Table 3.8: Comparison of decision trees for NI, RD, and TALOS

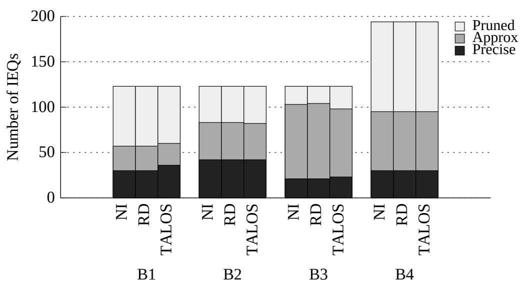
rithms using the MDL and F-measure metrics, respectively. The results show that `TALOS` produces much better quality IEQs than both `NI` and `RD`: while the average value of the MDL metric for `TALOS` is low (under 700), the corresponding values of both `NI` and `RD` are in the range of [4000, 22000]. For the F-measure metric, the average value for `TALOS` is no smaller than 0.7, whereas the values for `NI` and `RD` are only around 0.4.

Figure 3-6 compares the three algorithms for the three sets of workload queries, W_1, W_2, and W_3, on the Adult data set. As the results in Figure 3-6(a) show, `TALOS` again outperforms both `NI` and `RD` in terms of the running time. For workloads W_1 and W_2, the results in Figures 3-6(b) and (c) show that `TALOS` is able to find many more precise IEQs for all queries compared to `NI` and `RD`. The reason for this is that such queries have a larger number of free tuples per one partition, which gives `TALOS` more flexibility to derive precise IEQs. Figure 3-6(d) shows the comparison for the query workload W_3. As the average number of free tuples per one partition is smaller for queries in W_3, the flexibility for `TALOS` becomes reduced; however, `TALOS` still obtains about 1.5 to 9 times larger number of precise IEQs compared to `NI` and `RD`.

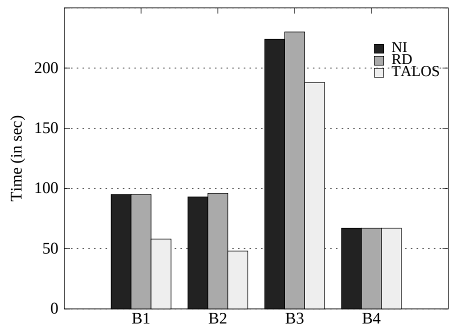
Figure 3-7 shows the comparison results for the Baseball data set for strong IEQs [13]. The results also demonstrate similar trends with `TALOS` outperforming `NI` and `RD` in both the running time as well as the number and the quality of IEQs generated for queries B_1 to B_3. It happens that all the tuples are bound for query B_4; hence, the performance results are the same for all three algorithms.

We observe that the benefit of `TALOS` over `NI` and `RD` is higher for queries A_1 - A_3 in Adult data set than that for queries B_1 - B_3 in Baseball data set. For example, `TALOS`

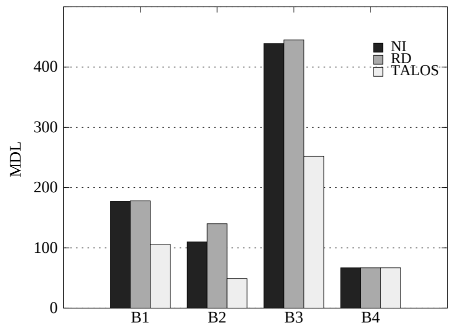
[13]The strong IEQs for the queries B_1 to B_3 actually turn out to be weak IEQs as well.



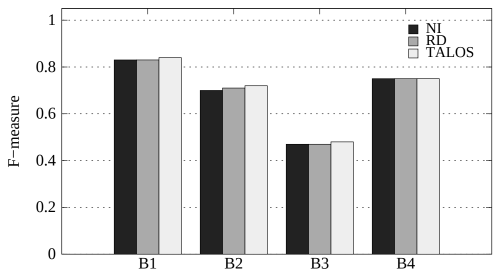
(a) Number of IEQs



(b) The running time



(c) MDL metric



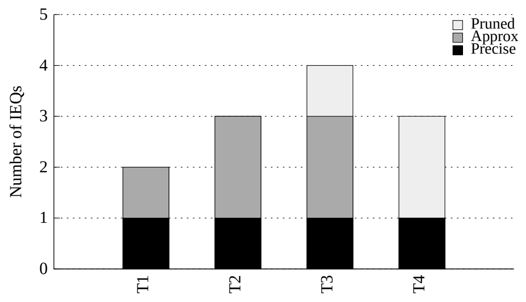
(d) F-measure metric

Figure 3-7: Comparison of TALOS, NI and RD for queries in Baseball

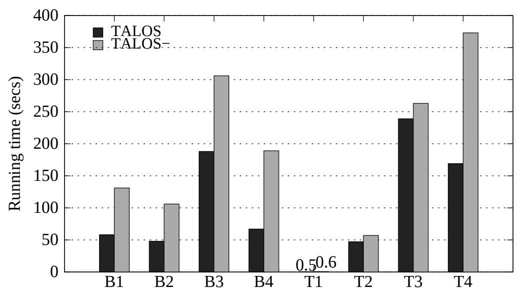
runs 3 - 10 times faster than `NI` and `RD` for queries A_1 - A_3, whereas `TALOS` runs 1.2 - 1.5 times faster than `NI` and `RD` for queries B_1 - B_3. As another example, the MDLs of the IEQs for A_1 - A_3 returned by `TALOS` are 10 - 1000 times lower than those of `NI` (and `RD`); whereas the MDLs of the IEQs for B_1 - B_3 returned by `TALOS` are two times lower than those of `NI` (and `RD`). The reason is that the number of free tuples per one partition for queries in Baseball data set is smaller (i.e., in the order of 10) than that of Adult's queries (i.e., in the order of 100). The flexibility for `TALOS`, therefore, reduces in queries B_1 - B_3; however, `TALOS` still produces higher quality IEQs than `NI` and `RD`.

3.6.4 Effectiveness of Optimizations

Figure 3-8(a) shows the number of strong IEQs produced by `TALOS` for test queries in TPCH data set; and Figure 3-8(b) examines the effectiveness of the optimizations by comparing the running times of `TALOS` and `TALOS`$^-$ on both the Baseball and TPCH data sets. Note that the number and quality of the IEQs produced by `TALOS` and `TALOS`$^-$ are



(a) Number of IEQs in TPCH queries



(b) The running time

Figure 3-8: Optimization of TALOS

Step	T_3		T_4	
	TALOS	TALOS$^-$	TALOS	TALOS$^-$
Join relation	41	61	22	106
Decision tree	198	202	147	267

Table 3.9: Detailed running times of TALOS and TALOS$^-$ (in seconds)

the same as these qualities are independent of the optimizations. The results show that TALOS is about 1.1 - 2 times faster than TALOS$^-$. The reason is that the computation of the hub table J_{hub} by TALOS using join indices is more efficient than the computation of the join relation J by joining relations directly in TALOS$^-$. In addition, the attribute lists accessed by TALOS, which correspond to the base relations, are smaller than the attribute lists accessed by TALOS$^-$, which are based on J.

To illustrate the observations above, we analyze the running time comparisons between TALOS and TALOS$^-$ to derive the IEQs for queries T_3 and T_4 with respect to two main steps of each algorithm including: (1) deriving the join relation, and (2) building decision trees. The results in Table 3.9 clearly demonstrate the effectiveness of TALOS over TALOS$^-$ in these steps. For instance, the step to derive the join relation for T_3 by TALOS is 1.5 times slower than that of TALOS$^-$, since TALOS only needs to join the corresponding join dices of *part*-*partsupp* and *partsupp*-*supplier* consisting of integer-valued columns. In contrast, TALOS$^-$ needs to perform the join among *partsupp*, *part*, and *supplier* relations. In another example, the step to derive the join relation for T_4 by TALOS is 4.5 times slower than that of TALOS$^-$, since TALOS only needs to read the corresponding join index (*lineitem-order*); whereas TALOS$^-$ needs to perform the join between *lineitem* and *order* relations. The attribute lists used by TALOS are also more compact than these used by TALOS$^-$; for instance, the attribute list constructed by TALOS$^-$ for attribute "*O.orderstatus*" for query T_4 is 4 times larger than that constructed by TALOS. This fact helps the steps to build decision trees in TALOS run more efficiently than TALOS$^-$.

Storage Overhead. The storage overhead of TALOS consists of pre-computed join indices built for pairs of relations that have foreign-key relationships. Basically, for every pair of relations R and S that have foreign-key relationship, TALOS builds a join index relation $I_{R,S}(r_R, r_S)$ consisting of k pairs of integers, where k is the number of tuples in the join result of R and S. In addition, TALOS also builds two B$^+$-indices on r_R and r_S column of $I_{R,S}$ to speed up the joins of using join indices. In our experiments, the pre-computed join indices built for TPCH data set consist of 460MB versus 1GB of the whole database. Correspondingly, the join indices for Baseball data set consist of 16MB over 35MB size of the database. Note that we do not build any join indices for Adult data set, since Adult includes only a single relation.

Average Time to Derive One IEQ. For queries B_1 - B_4 on the Baseball data set, the number of IEQs (both precise and approximate) generated by TALOS is in the range $[60, 100]$ with an average running time of about 100 seconds. Thus, it takes TALOS about one second to generate one IEQ, which is reasonable. For the queries T_1 - T_4 on TPCH data set, TALOS returns one IEQ for T_1 - T_4 in averagely 0.3, 22, 60 and 56 seconds, respectively. Overall, even for the large TPCH data set, the running time for TALOS is still reasonable.

3.6.5 Strong and Weak IEQs

In this section, we discuss some of the IEQs generated by TALOS for the various queries. The samples of weak and strong IEQs generated from Adult data set are shown in Ta-

Q	IEQ	$\lvert p_a \rvert$	$\lvert p_b \rvert$	$\lvert p_c \rvert$	F_m^{est}
$A_{1,1}$	$\sigma_{gain>7298 \wedge ms=\text{“Married-AF”}}\ (adult)$	1	0	0	0.63
$A_{1,2}$	$\sigma_{edu=\text{“Preschool”} \wedge race=\text{“Eskimo”}}\ (adult)$	1	0	0	0.25
$A_{1,3}$	$\sigma_{loss>3770}\ (adult)$	1	0	0	0.24
$A_{2,1}$	$\sigma_{(age\leq 85 \wedge hpw\leq 1 \wedge edu>13)\vee(age>85 \wedge edu=\text{“Master”} \wedge hpw\leq 40)}\ (adult)$	4	0	0	0.004
$A_{3,1}$	$\sigma_{sex=\text{“Female”}}\ (adult)$	111	78	26	-

Table 3.10: Weak IEQs on Adult

Q	IEQ	$\lvert p_a \rvert$	$\lvert p_b \rvert$	$\lvert p_c \rvert$
$A_{1,4}$	$\sigma_{p_1 \wedge p_2}\ (adult)$ $p_1 : 48 < hpw \leq 50 \wedge race \notin \{\text{“Eskimo”},\text{“Asian”}\}$ $p_2 : 6849 < gain \leq 7298 \wedge loss \leq 0 \wedge edu\_num > 14$	1	1	13
$A_{2,2}$	$\sigma_{(63<age\leq 66 \wedge edu>15 \wedge ms=\text{“NM”})\vee(66<age\leq 68 \wedge ms=\text{“NM”} \wedge gain>2993)}\ (adult)$	5	0	0
$A_{3,2}$	$\sigma_{ms=\text{“Never married”}}\ (adult)$	15000	0	0
$A_{4,1}$	$\sigma_{(1055<gain\leq 27828 \wedge age\leq 17)\vee(gain>27828 \wedge occ=\text{P} \wedge race\neq\text{O})}\ (adult)$	4	0	0

Table 3.11: Strong IEQs on Adult

bles 3.10 and 3.11, respectively. Table 3.12 shows sample strong IEQs generated from Baseball and TPCH data sets[14]. For each IEQ, we also show its value for the F-measure or F_m^{est} metric. In Tables 3.10 - 3.12, the F-measure metric values are shown in terms of their $|p_a|$, $|p_b|$ and $|p_c|$ values; an IEQ is precise iff $|p_b| = 0$ and $|p_c| = 0$. We use $X_{i,j}$ to denote an IEQ for a query X_i, $X \in \{A, B, T\}$.

Adult. In query A_1, we want to know the native country of people whose occupation is in the Armed Force. The query result is “U.S”. From the weak IEQs, we learn that the people who are married to someone in the Armed Force and have high capital gain ($A_{1,1}$) have the same native country “U.S”; or people with high capital loss (> 3770) also have “U.S” as their native country ($A_{1,3}$).

In query A_2, we want to find the occupation and education of white females who are never married with age in the range $[64, 68]$, and have capital gain > 500. The strong IEQ $A_{2,2}$ provides more insights about this group of people: those in the age range $[64, 66]$ are highly educated, whereas the others in the age range $[67, 68]$ have high capital gains.

In query A_3, we want to know the native country and capital gain of people who have

[14]The strong IEQs shown in Table 3.12 actually turn out to be strong IEQs as well for the queries B_1 to B_3.

marital status as "never-married". The query selects 15000 people and has 137 distinct pairs of the attribute values of (native country, capital gain) in the query result. The weak IEQ $A_{3,1}$ shows that all the females (in the data set) have their native country and capital gain attribute values cover 111 over 137 tuples the query result of A_3.

Query A_4 is a skyline query looking for people with maximal capital gain and minimal age. Both strong and weak IEQs return the same IEQs for this query. Interestingly, the precise IEQ $A_{4,1}$ provides a simplification of A_4: the people selected by this skyline query are either (1) very young ($age \leq 17$) and have capital gain in the range $1055 - 27828$, or (2) have very high capital gain (> 27828), work in the protective service, and whose race is classified as "others".

Baseball. In query B_1, we want to find all players who belong to team "ARI" in 2006 and have a high performance ($HR > 10$). The result includes 7 players. From the IEQ $B_{1,1}$, we know more information about these players' performance (G, RBI, etc.), and their personal information (e.g., birth year). In addition, from IEQ $B_{1,2}$, we also know that one player in this group got an award when he played in "NL" league.

In query B_2, we want to find the set of high performance players who have very high total home runs (> 600). The IEQ $B_{2,1}$ indicates that some of these players play for "NY1" team. The IEQ $B_{2,2}$ indicates one player in this group is very highly paid and has a left throwing hand.

Query B_3 is a skyline query that looks for players with maximal number of home runs (HR) and minimal number of strike outs (SO). The result has 35 players. The IEQs provide different characterizations of these players. Query $B_{3,1}$ indicates that two players in this group are also the managers of "WS2" and "NYA" teams; while query $B_{3,2}$ indicates that two players in this group are averagely paid.

Query B_4 is an interesting query that involves multiple core relations. This query asks for the managers of team "CIN" from 1983 to 1988, the year they managed the team as well as the rank that the team gained. There are 3 managers in the result. In this query, we note that `TALOS` found alternative join-paths to link the two core relations, Manager

Q	IEQ	$\lvert p_a\rvert$	$\lvert p_b\rvert$	$\lvert p_c\rvert$
$B_{1,1}$	$\sigma_{p_1 \vee p_2}$ $(Master \bowtie Batting)$ $p_1 : (team = \text{“ARI”} \wedge G \leq 156 \wedge 70 < RBI \leq 79 \wedge year > 1975)$ $p_2 : (team = \text{“ARI”} \wedge G > 156 \wedge BB \leq 78)$	7	0	0
$B_{1,2}$	$\sigma_{lg=\text{“NL”} \wedge month=12 \wedge 71<height \leq 72 \wedge nc \neq \text{“D.R”}}$ $(Master \bowtie AwardsPlayer)$	1	0	6
$B_{2,1}$	$\sigma_{p_1 \vee p_2 \vee p_3}$ $(Master \bowtie Batting)$ $p_1 : (BB \leq 162 \wedge HR > 46 \wedge birthCity = \text{“Mobile”} \wedge RBI \leq 127)$ $p_2 : (BB \leq 162 \wedge HR > 46 \wedge teamID = \text{“NY1”} \wedge BB > 74)$ $p_3 : (BB > 162)$	4	0	0
$B_{2,2}$	$\sigma_{salary>21680700 \wedge throws=\text{“L”}}$ $(Master \bowtie Salaries)$	1	0	3
$B_{3,1}$	$\sigma_{(team=\text{“WS2”} \wedge R \leq 4) \vee (team=\text{“NYA”} \wedge state=\text{“LA”})}$ $(Master \bowtie Manager)$	2	0	33
$B_{3,2}$	$\sigma_{(p_1 \vee p_2)}$ $(Master \bowtie Salaries)$ $p_1 : (height \leq 78 \wedge weight > 229 \wedge country = \text{“DR”} \wedge 180000 < salary < 195000)$ $p_2 : (height > 78 \wedge state = \text{“GA”} \wedge salary > 302500)$	2	0	33
$B_{4,1}$	$\sigma_{21<L \leq 22 \wedge SB \leq 0 \wedge 67<W \leq 70}$ $(Mananger \bowtie Master \bowtie Batting \bowtie Team)$	1	0	5
$T_{1,1}$	$\sigma_{(N.regionakey \leq 3)}$ $(supplier \bowtie nation)$	4383	3598	0
$T_{2,1}$	$(customer \bowtie nation)$	95264	54736	0
$T_{3,1}$	$\sigma_{3000<PS.availqty \leq 3001 \wedge P.retailprice \leq 953}$ $(part \bowtie partsupp \bowtie supplier)$	1	0	24671
$T_{4,1}$	$\pi_{O.clerk,L.extendedprice}$ $\sigma_{L.quantity \leq 1 \wedge orderstatus=\text{“P”}}$ $(lineitem \bowtie order)$	3719	0	0

Table 3.12: Strong IEQs on Baseball and TPCH

and Team. The first alternative join-path (shown by $B_{4,1}$) involves Manager, Master, Batting, and Team. The second alternative join-paths (not shown) involves Manager, Master, Fielding, and Team. The IEQ $B_{4,1}$ reveals the interesting observation that there is one manger who was also a player in the same year that he managed the team with some additional information about this manager-player.

3.7 Summary

In this chapter, we have described our proposed solution, TALOS, that models the problem to derive instance-equivalent queries as a data classification task with a unique property that we term *at-least-one semantics*, which is inherent in the derivation of IEQs. To handle data classification with this new semantics, we developed a new dynamic class labeling technique. In addition to the basic framework, we designed several optimization techniques to reduce processing overhead, including join indices and mapping tables. Furthermore, as there can be multiple IEQs, we introduced a set of criteria to rank order output queries by various notions of utility, including the minimum description length and F-measure. Our experimental evaluation of TALOS demonstrates its efficiency and effectiveness in generating interesting IEQs.

Chapter 4

REQUERE: Reverse-Query Engineering System

In the previous chapter, we have described the framework of TALOS to derive Select-Project-Join relational query Q' that is instance-equivalent to a given input query Q w.r.t. a single database version D; i.e., $Q'(D) = Q(D)$. Such queries can shed light on hidden relationships within the data, provide useful information on the relational schema, as well as potentially summarize the original query.

In this chapter, we present a generalized framework of TALOS, named REQUERE for **Re**verse **Quer**y **E**ngineering, that generalizes TALOS along three key dimensions of the problem setting including (1) the original query Q (i.e., Q can be unknown), (2) the database version D (i.e., there have multiple database versions), and (3) the derived query Q' (i.e., Q' is in more expressive fragments). These generalizations are important to broaden the range of applications of QBO.

4.1 An Overview of REQUERE

The generalization of REQUERE over TALOS is summarized in Table 4.1. We discuss these generalizations in details next.

Parameter	**TALOS**	**REQUERE**
Database	Single version D	Multiple versions $D_1, D_2, \cdots, D_k$
Original Query Q	Q is known	Q is unknown Case 1: $Q(D_i)$ is known for *some* $i \in [1, k]$ Case 2: $Q(D_i)$ is known for *each* $i \in [1, k]$
Derived Query Q'	Q' is a SPJ query	Q' is a SPJ, SPJU, or SPJA query

Table 4.1: Summary of the Generalization of REQUERE over TALOS

4.1.1 Unknown Query

First, unlike TALOS, REQUERE also considers the scenarios where the input to the problem consists of only a given result table $T = Q(D)$ but not the original Q itself. The given result table T in these contexts may have been derived manually (e.g., a user selected some tuples of the database of interest to her), or by an application program that is no longer available (e.g., the program is no longer maintained or is lost). Such scenarios are more common in data exploration, where the documentation and meta-data for the data sets being analyzed are incomplete, inaccurate, or missing (e.g., AT&T's Bellman project [24]).

The absence of the input query Q makes it more challenging to identify the core relations to be included in the instance-equivalent query Q' of the given result table T. REQUERE makes use of *domain indices* to efficiently identify core relations for the given result table T.

4.1.2 Multiple Database Versions

Second, in contrast to TALOS, which solves the problem where the specific database D is known and given as part of the input, the setting considered by REQUERE is often more general where there could have multiple database versions. We consider two specific scenarios of this generalization and the additional challenges introduced by them.

In the first data exploratory/analysis scenario, a user might need to reverse-engineer a query Q' from a result table T that was generated some time ago by some unknown query Q. Thus, it may not be meaningful or possible to derive Q' from the current version of the database, as this could be very different from the version that T was generated from. Specifically, given a result table T and a sequence of database versions $< D_1, D_2, \cdots, D_\ell >$, a specific goal may be to determine the most recent database version D_i and an IEQ Q' such that $Q'(D_i) = T$. Depending on the applications, other variations of the problem (e.g., finding the earliest database version or all versions) are also possible. The performance challenge is how to determine both D_i as well as Q' for a given result table T efficiently.

In the second data analysis scenario, the user is provided with more information in the form of a sequence of database versions and result pairs (D_1, T_1), (D_2, T_2), $\cdots$, (D_ℓ, T_ℓ); where each T_i is the result of *the same* unknown query Q on database version D_i (i.e., $T_i = Q(D_i)$). For example, the T_i's could correspond to weekly reports generated by the same query on weekly versions of the database, or Q could be a continuous long standing query that is run periodically on different snapshots of the database. In this more general setting with multiple database and result versions, the challenge is to efficiently reverse-engineer a query Q' such that $Q'(D_i) = T_i$ for each $i \in [1, \ell]$. REQUERE introduces a new labeling scheme to solve this problem setting efficiently.

4.1.3 Supporting More Expressive IEQs

Third, while TALOS derives IEQs that belong to the simple fragment of Select-Project-Join (SPJ) relational queries, REQUERE is designed to be able to handle more expressive classes of queries beyond SPJ-queries. REQUERE can support not only SPJ-IEQs, but also SPJ-IEQs with union operators (referred to as *SPJU-IEQs*), and SPJ-IEQs with group-by aggregation operators (referred to as *SPJA-IEQs*). With this enhanced expressiveness, REQUERE becomes useful in more application domains such as data integration, where SPJU-IEQs are predominant. In data integration systems, the goal is to combine data residing at different sources to provide users with a unified view of these data [30]. The *global-as-view* integration approach requires that the global schema be expressed in terms of the data sources, which necessitates a query over the global schema to be reformulated in terms of a set of queries over the data sources. Thus, the QBO problem in this context is: Given a result table T that is generated by the integration system, derive the query that is a union of sub-queries over the data sources. Another application domain that is supported by the enhanced expressiveness of IEQs is in data analysis, where SPJA-IEQs are very common due to aggregation computations (e.g. group-by aggregation queries in OLAP applications).

For efficiency reasons, the default mode of operation for REQUERE is first to try to derive IEQs that belong to a simpler fragment before proceeding to the more complex fragments. Specifically, REQUERE derives IEQs using the following sequence of query fragments: SPJ, SPJU, and SPJA. Thus, given a result table $T = Q(D)$, REQUERE will first attempt to derive an IEQ Q' that is an SPJ query. If such an IEQ is found, REQUERE will return this IEQ and terminate; otherwise, REQUERE will proceed to derive an SPJU-IEQ, and so on. However, depending on the user's preference or application need, REQUERE can easily reorder this default fragment sequence. As an example, if a user has prior-knowledge that a result table $Q(D)$ has at least one aggregated attribute, then she may want REQUERE to consider only SPJA-IEQs.

Organization. The remaining of this chapter is organized as follows. Section 4.2 presents the preliminaries for REQUERE. Section 4.3 introduces the proposed domain indices technique that REQUERE adopts to solve the issue of unknown input query. Sections 4.4 and 4.5 discuss how REQUERE handles multiple database versions. Section 4.6 presents the techniques of REQUERE to derive the IEQs in more expressive fragments. Section 4.7 presents an experimental evaluation of REQUERE. Finally, Section 4.8 summarizes our work on REQUERE.

4.2 Preliminaries

In this work, we consider three fragments of SPJ relational queries for IEQs, where each projected attribute is either some relation's attribute, or a value computed by an aggregation operator (COUNT, SUM, or AVG) that does not involve any arithmetic expression in the operator's argument. Specifically, SPJ queries are the basic select-project-join queries. SPJU queries are of the form Q_1 *union* Q_2 $\cdots$ *union* Q_n, where each Q_i is an SPJ query. SPJA queries correspond to simple SPJ SQL queries with aggregation operators in the select-clause and an optional group-by clause.

We use $\mathcal{SG}$ to denote the schema graph of a database D. Each node in $\mathcal{SG}$ represents a relation in D, and an edge in $\mathcal{SG}$ represents a foreign-key join between a pair of relations associated with the two connected nodes. We refer to the attributes in the database schema as *schema attributes*.

In the rest of the discussions, we focus on finding *precise* instance-equivalent queries (IEQs) Q' for a given result table T; i.e., $Q'(D)$ is exactly equal to T. The techniques to derive *approximate* IEQs Q' (i.e., $Q'(D)$ differs from T in some tuples) require minor modifications and are omitted here.

4.2.1 Review of TALOS

Since `REQUERE` is built on `TALOS`, we first review `TALOS` that is designed for the setting of `QBO` where the original query Q is known with a single database version D and the derived IEQs are limited to simple SPJ queries. The procedure of `TALOS` is sketched in Algorithm 2.

Algorithm 2: `TALOS` – *basic*(Q, D)

```
1 Compute sets of core relations of $Q$;
2 foreach set of core relation $\mathcal{R}$ do
3     Enumerate schema subgraphs $G$ containing $\mathcal{R}$;
4     foreach schema subgraph $G$ do
5         Compute the join relation $J(G)$;
6         Build decision trees on $J(G)$;
7     end
8 end
```

To generate an SPJ query Q' that is instance-equivalent to an input query Q, `TALOS` basically needs to determine the three components of Q': $proj(Q')$, $rel(Q')$, and $sel(Q')$.

The first two components are easily derived when Q is given; i.e., TALOS requires Q' to include all the projected attributes of Q in its select-clause. Therefore, Q' must at least contain a minimal set of relations, called the set of *core relations* (denoted as $\mathcal{R}$), such that every projected attribute of Q belongs to one relation in $\mathcal{R}$ (line 1). Note that there could have more than one sets of core relations for a given query Q. For example, consider a query Q: $\pi_{S.A}\sigma_{S.A=T.B}(S \times T)$; there are two sets of core relations for Q including $\mathcal{R}_1 = \{S\}$ and $\mathcal{R}_2 = \{T\}$.

For each computed set of core relations $\mathcal{R}$, TALOS considers different schema subgraphs G of $\mathcal{SG}$, each of which contains all relations in $\mathcal{R}$ (line 3); thus, G determines both the relations that appear in the from-clause of Q' as well as the foreign-key join predicates that appear in the where-clause of Q'. The main challenge for TALOS is deriving the selection predicates for the IEQ Q'. The approach adopted by TALOS is to model the problem as a data classification task, which is solved by constructing different decision trees to generate different sets of selection predicates and hence different IEQs (w.r.t. G) for Q as follows.

Conceptually, TALOS computes a join relation $J(G)$ by joining all the relations in G based on the foreign-key joins represented in G (line 5). To build the decision trees on $J(G)$ (line 6), TALOS partitions $J(G)$ into two disjoint subsets $J(G) = J_0 \cup J_1$ such that: $\pi_{proj(Q)}(J_1) = Q(D)$, and $\pi_{proj(Q)}(J_0) \cap Q(D) = \emptyset$. For the purpose of deriving $sel(Q')$, TALOS labels the tuples in J_0, which do not contribute to the query result $Q(D)$, as *negative tuples*. TALOS labels a subset $J_1' \subseteq J_1$ as *positive tuples* (with tuples in $J_1 - J_1'$ as negative) such that: (1) $\pi_{proj(Q)}(J_1') = \pi_{proj(Q)}(J_1)$ (without affecting the imprecision of Q'), and (2) $sel(Q')$ is succinct (without too many conditions). The first condition is due to the fact that multiple tuples in J_1 can be projected to the same tuple in $\pi_{proj(Q)}(J_1)$. Given the labeled tuples in $J(G)$, the problem of finding a $sel(Q')$ can now be viewed as a data classification task to separate the positive and negative tuples in $J(G)$: $sel(Q')$ is given by the selection conditions derived from the decision tree built to specify the positive tuples. We will not go into details on how TALOS derives J_1', as it does not affect our discussions for REQUERE.

There are some optimization techniques introduced in TALOS; for instance, since the computational for $J(G)$ is costly, TALOS optimizes the performance by actually not computing $J(G)$ as described. We will also not go into details of TALOS's optimizations, as it will not affect our discussions for REQUERE.

4.2.2 Multiple Database Version Organization

Multiple database versions are typically organized using a *reference version* (either the earliest or the latest version) together with a sequence of forward/backward deltas between successive versions (e.g., [45]). In this work, without loss of generality, we assume the "backward" delta storage organization; i.e., given a sequence of database versions $D_1, D_2, \cdots, D_\ell$, the database stores the most recent version D_ℓ together with $\delta_{\ell(\ell-1)}, \cdots, \delta_{21}$; where each D_j can be derived from D_i and δ_{ij}. For simplicity, we assume that each tuple update operation is modeled by a pair of tuple delete and insert operations. Thus, each delta δ_{ij} consists of a set of tuple insert and delete operations.

4.3 Unknown Query

This section addresses the first challenge of REQUERE to derive SPJ-IEQs Q' given a specific database D and a result table $T = Q(D)$ without the knowledge of Q. The key issue is how to efficiently determine sets of core relations of T (corresponding to the first step of TALOS in Algorithm 2-line 1). REQUERE introduces a simple but effective indexing technique, called *domain indices*, to solve this issue.

Essentially, for each column C in the given result table T, REQUERE needs to determine some schema attribute A_i that can "completely cover" C in the sense that the column of attribute values for A_i contains all values in the column C. We refer to a schema attribute A_i that completely covers a column C as a *matching attribute* of C. Once REQUERE has determined a set S of matching attributes, each of which completely covers a different column of T such that all columns of T are completely covered, REQUERE considers S as a set of projected attributes of Q' (i.e., $proj(Q') = S$). The set of core relations $\mathcal{R}$ for Q' corresponding to S is given by the set of relations in D containing attributes in S. REQUERE then executes the remaining steps of TALOS to derive the IEQs for T w.r.t. the set of core relations $\mathcal{R}$, as described in Algorithm 2.

We note that for some value v that appears more than one time in a column C, REQUERE only requires the column of a matching attribute A_j of C to contain v at least one time. The reason is that after joining all relations in the derived subgraph G containing $\mathcal{R}$, the A_j's column in the join relation $J(G)$ might contain enough the number of instances of v. For simplicity and without loss of generality, we assume that each column C of T does not contain any duplicate values from now on; otherwise, we simply pre-process C to eliminate duplicates.

4.3.1 Naive solution

To find matching attributes of a column C of data type d (e.g., categorical, numerical) in T, a straightforward solution is to intersect C with the column of each schema attribute A_j of the same data type d one at a time. If the intersection result between A_j and C contains all values in C, then A_j is a matching attribute of C. We can optimize this process further by intersecting the column of each schema attribute A_j with a single "domain" column $\mathcal{CB}$, which is the result of merging all columns C in T that have the same data type as A_j, instead of intersecting A_j with each column C separately. We refer to this naive solution as REQUERE$^-$.

Example 4.1 *Consider a given result table T_1 in Figure 4-1(a) consisting of two columns C_1 and C_2. To reverse-engineer an IEQ Q' of T_1,* REQUERE$^-$ *first finds matching attributes for C_1 and C_2.* REQUERE$^-$ *intersects C_1 with the column of each categorical attribute in D, and observes that only Master.name's column contains all distinct values in C_1. Thus, Master.name is the matching attribute of C_1. In a similar way,* REQUERE$^-$

C_1	C_2
C	35
E	60

(a) Given result table T_1

Column	Matching attributes
C_1	{*Master*.name}
C_2	{*Salaries*.salary, *Batting*.HR}

(b) Matching attributes

Set of matching attributes	Set of core relations
$S_1 = \{Master.name, Salaries.salary\}$	$\mathcal{R}_1 = \{Master, Salaries\}$
$S_2 = \{Master.name, Batting.HR\}$	$\mathcal{R}_2 = \{Master, Batting\}$

(c) Core relations

Figure 4-1: Example of finding IEQs on single database version

intersects column C_2 with the column of each numerical attribute in D, and derives that Salaries.salary and Batting.HR are the two matching attributes of C_2.

Therefore, there are two sets of matching attributes S_1 and S_2, and correspondingly two sets of core relations $\mathcal{R}_1$ and $\mathcal{R}_2$ (shown in Figures 4-1(b) and (c)). With $\mathcal{R}_1$, REQUERE$^-$ *derives an IEQ $Q'_{1,1} = \pi_{name,salary}\ \sigma_{bats=\text{“}R\text{”}}\ (Master \bowtie Salaries)$. Similarly, with $\mathcal{R}_2$,* REQUERE$^-$ *derives another IEQ $Q'_{1,2} = \pi_{name,HR}\ \sigma_{bats=\text{“}R\text{”} \wedge stint>1}\ (Master \bowtie Batting)$.*

□

4.3.2 Domain indices

REQUERE optimizes the process of finding matching attributes by using a simple but yet effective indexing technique, called *domain indices*. Unlike a conventional index that is defined on attribute(s) within a single relation, a domain index is defined on all the attributes in the database that have the same attribute domain. By indexing on a database domain instead of a relation attribute, domain indices enable matching attributes to be determined efficiently.

For each data type d (e.g., categorical, numerical), REQUERE maintains a three-column mapping table $M_d(v, attr, count_v)$, where v is a value of type d in the database, *attr* is the schema attribute that contains v in its column, and $count_v$ is the number of times that v appears in *attr*'s column. This table has one composite key consisting of v and *attr*, which is indexed by a B$^+$-tree that we refer to as a *domain index*. When database

v	attr	count$_v$
A	Master.name	1
C	Master.name	1
E	Master.name	1
P1	Batting.pID	2
P1	Salaries.pID	1
...	...	...

(a) Categorical attributes

v	attr	count$_v$
35	*Salaries*.salary	1
60	*Salaries*.salary	1
35	*Batting*.HR	1
60	*Batting*.HR	1
1	Batting.stint	2
...	...	...

(b) Numeric attributes

Figure 4-2: The mapping tables for Baseball database

records are modified, $count_v$ is updated accordingly. Whenever $count_v$ is 0, the corresponding value v will be removed from the table M_d and its domain index. The column "$count_v$" is mainly used to facilitate index maintenance.

Conceptually, to determine the matching attributes for a column C of data type d in T, REQUERE joins C with the mapping table of the same data type (M_d) using the domain index. The join result is a relation $\mathcal{RM}(v, attr)$. REQUERE further performs a group-by aggregation on $\mathcal{RM}$ to derive a relation $\mathcal{RM}_g(attr, num\_matching)$ with $num\_matching$ is derived by applying a COUNT operation on v's column. Each row (A, num) of $\mathcal{RM}_g$ indicates the number of distinct values (i.e., num) in the column of C that are contained in the column of the attribute A. Therefore, a schema attribute A is a matching attribute of C if (A, num) is a tuple in $\mathcal{RM}_g$ with num is equal to the number of distinct values in C.

Similar to REQUERE$^-$, REQUERE also further optimizes this process by joining a column $\mathcal{CB}$, which is the result of merging all columns C in T of the same data type d, with the mapping table of type d (M_d) one at the time instead of joining each C with M_d separately.

Example 4.2 *We illustrate how* REQUERE *uses domain indices to find matching attributes for columns C_1 and C_2 of T_1 in Example 4.1. The mapping tables built for categorical and numeric attributes are shown in Figures 4-2(a) and (b).* REQUERE *joins column C_1 with the mapping table of categorical attributes, and derives a relation $\mathcal{RM}$ = {(C,Master.name), (E,Master.name)}.* REQUERE *further groups $\mathcal{RM}$ on $\mathcal{RM}$.attr to de-*

```
SELECT     M_d.attr, CT.cid, count(M_d.v) AS num_matching
FROM       M_d, CT
WHERE      M_d.v = CT.v
GROUP BY   M_d.attr, CT.cid
```

Figure 4-3: The SQL query to derive matching attributes

rive $\mathcal{RM}_g$ = {*(Master.name,* 2)}. *Base on the resultant relation* $\mathcal{RM}_g$*,* REQUERE *concludes that Master.name is a matching attribute of* C_1 *since the number of distinct values in* C_1 *is also* 2*. In a similar way,* REQUERE *derives Salaries.salary and Batting.HR as the matching attributes of* C_2*.* □

Implementation Details. For efficiency, REQUERE stores *attr* as an integer value representing the schema attribute's identifier. Furthermore, REQUERE does not compute $\mathcal{RM}$ as described. Instead, REQUERE derives a temporary relation $CT(v, cid)$, where v is a value of data type d appeared in T and cid is the identifier of the column in T that contains v. Computing $\mathcal{RM}$ and deriving the matching attributes are implemented by executing the SQL query in Figure 4-3. From the result of this query, a schema attribute corresponding to "$M_d.attr$" is a matching attribute of the column corresponding to "$CT.cid$" in T if *num_matching* is equal to the number of values in $CT.cid$'s column.

4.4 Multiple Database Versions & Single Unknown Result

This section explains how REQUERE derives IEQs in the presence of multiple database versions and a single unknown result, where the goal is to derive IEQs w.r.t. the most recent possible database version. The inputs to the problem considered in this section are a given result table $T = Q(D)$ and a sequence of ℓ database versions organized in the form D_ℓ, $D_{\ell-1}, \cdots, D_1$; where $D_i = D_{i+1} \oplus \delta_{(i+1)i}$ for $i \in [1, \ell - 1]$ and D_ℓ is the current database version. Here, we use the notation "$X \oplus \delta X$" to denote applying the insert/delete "delta" tuples in δX to update X, where X is a database version or a relation. The goal is to efficiently identify the most recent database version D_i such that there exists an IEQ Q' with $Q'(D_i) = T$.

The most straightforward solution is to apply the previous solution developed for a single database version by trying to find the IEQs starting from the most recent database version, and progressively working "backwards" to the next recent version and so on until an IEQ is derived (or none is found). For each database version D_i considered, REQUERE basically identifies sets of matching attributes S_i for T, followed by sets of core relations $\mathcal{R}$ corresponding to S_i (Algorithm 2-line 1), and a schema subgraph G containing $\mathcal{R}$. After deriving G, REQUERE joins all the relations in G using the foreign-key joins in G to derive a relation $J_i(G)$ (Algorithm 2-line 5), and then derives the IEQs for T w.r.t. $J_i(G)$.

In the following discussions, we present how REQUERE optimizes the computation of S_i and $J_i(G)$. The main ideas of REQUERE are to utilize the computations for S_i and $J_i(G)$ in the previously considered version to the current database version being considered. Our basic techniques are based on the well-known join view maintenance techniques [5, 37].

4.4.1 Optimizing Matching Attributes Computation

Suppose that REQUERE has already computed the matching attributes for database version D_{x+1}, and is currently considering database version D_x where $D_x = D_{x+1} \oplus \delta_{(x+1)x}$. To simplify the presentation, we discuss how REQUERE finds the matching attributes for numerical columns of T; the same principles are applied for columns in T of different data types (e.g., categorical) as well. Similar to the discussions in Section 4.3, we also preprocess each column C in T so that C does not contain any duplicate values.

Let M_i denote the mapping table of numeric attributes for database version D_i, $i \in [1, \ell]$. Recall that to find the matching attributes for a numeric column C in T w.r.t. a database version D_{x+1}, REQUERE basically joins C with the mapping table M_{x+1} to derive a relation $\mathcal{RM}_{x+1}(v, attr, count_v)$, from which REQUERE can derive the matching attributes for C. Therefore, a straightforward solution to find matching attributes for C in the database version D_x is to derive the mapping table M_x (w.r.t. the database version D_x) by updating M_{x+1} with the inserted/deleted numeric values in $\delta_{(x+1)x}$, and then join C with M_x to derive a relation $\mathcal{RM}_x$ in the same role with $\mathcal{RM}_{x+1}$.

REQUERE optimizes this process by treating $\mathcal{RM}_x$ as a view of the join between M_x and C, where M_x is modified from M_{x+1}. There are two benefits of this optimization: (1) leveraging the join result between M_{x+1} and C for D_{x+1} (the result has already been stored in $\mathcal{RM}_{x+1}$), and (2) avoiding the costly operation of updating the mapping table M_{x+1}.

Algorithm 3: REQUERE$_M$

```
1 $\delta^n_{(x+1)x} \leftarrow \{(v, attr, c_a, c_r)\}$;
2 $M_n \leftarrow \emptyset$;
3 foreach $(v, attr, c_a, c_r) \in \delta^n_{(x+1)x}$ do
4     $M_n \leftarrow M_n \bigcup \{(v, attr, c_a - c_r)\}$;
5 end
6 $\mathcal{RM}_n \leftarrow M_n \bowtie_{M_n.v=C.v} C$;
7 $\mathcal{RM}_x \leftarrow \pi_{v,attr,\mathcal{RM}_{x+1}.count_v+\mathcal{RM}_n.count_v}(\mathcal{RM}_{x+1} ⟗_{v,attr} \mathcal{RM}_n)$;
```

We present the approaches of REQUERE to compute $\mathcal{RM}_x$ in Algorithm 3. Since $\delta_{(x+1)x}$ contains the set of inserted/deleted tuples into D_{x+1} to derive D_x, REQUERE first constructs a set $\delta^n_{(x+1)x}$ that will contain tuples in the form $(v, attr, c_a, c_r)$, where c_a (resp. c_r) represents the number of times that the corresponding numeric value v is inserted (resp. deleted) into (from) the column of the schema attribute $attr$. REQUERE then derives a relation $M_n(v, attr, count_v)$ that represents the "nett change" of the number of times that a value v is inserted into the column of $attr$ (line 4)[1]. Thus, M_x is equivalent to $M_{x+1} \oplus M_n$. It derives that the resultant relation $\mathcal{RM}_x$ of the join between M_x and C is equivalent to $(M_{x+1} \oplus M_n) \bowtie C$. The next two steps in lines 6 & 7 are then to compute $\mathcal{RM}_x$ using this formula. In line 7, we use the notion ⟗ to denote the full outer join operator.

Example 4.3 *Consider a sequence of two database versions: (1) the current version D_2, which is the baseball database given in Figure 3-1, and (2) the database version D_1,*

[1]If $count_v < 0$, it implies that v is deleted $(-count_v)$ times from $attr$'s column.

insert(P6, D, Korea, 75, R, R) into *Master*
insert (P6, 2004, 3, SFN, 60) into *Batting*
insert (P2, 2003, 1, PIT, 73) into *Batting*

(a) Delta δ_{21}

C_1	C_2
B	73
D	60

(b) Input table T_2

Figure 4-4: Finding IEQs on multiple database versions

which is given by the "delta" δ_{21} (shown in Figure 4-4(a)). The input table T_2, shown in Figure 4-4(b), has two columns C_1 and C_2.

To derive the IEQs for T_2, REQUERE *starts with the current database version D_2, and determines S_2 = {Master.name, Batting.HR} as the set of matching attributes of T_2 w.r.t. D_2.* REQUERE *then derives a schema subgraph G including one edge Master-Batting, computes $J_2(G)$ = Master $\bowtie$ Batting, and concludes that there does not exist any IEQs w.r.t. D_2, since $T_2 \not\subseteq \pi_{name,HR}(J_2(G))$.*

Therefore, REQUERE *needs to find IEQs of T_2 on D_1.* REQUERE *first derives the matching attributes for C_1 and C_2 w.r.t D_1. Using Algorithm 3,* REQUERE *derives a relation $M_n = \{(75, weight, 1), (60, HR, 1), (73, HR, 1), (1, stint, 1), (3, stint, 1)$,(2003, year, 1),(2004, year, 1)$\}$.* REQUERE *then derives $\mathcal{RM}_n = \{(60, HR, 1), (73, HR, 1)\}$. Note that when computing matching attributes on D_2,* REQUERE *has already computed $\mathcal{RM}_2 = \{(73, HR, 1), (60, HR, 1)\}$. Finally,* REQUERE *computes $\mathcal{RM}_1$ by performing a full outer join between $\mathcal{RM}_2$ and $\mathcal{RM}_n$ to derive $\mathcal{RM}_1 = \{(60, HR, 2), (73, HR, 2)\}$. Based on $\mathcal{RM}_1$,* REQUERE *concludes that Batting.HR is the matching attribute for C_2. In a similar way,* REQUERE *derives Master.name as the matching attribute for C_1.* □

4.4.2 Optimizing Join Relation Computation

We now explain how REQUERE efficiently derives the join relation $J_x(G)$ w.r.t. the database version D_x and a schema subgraph G, given that REQUERE has already derived the join relation $J_{x+1}(G)$ w.r.t. D_{x+1} and G.

One straightforward solution is to update the relations that are used to derive $J_x(G)$ with the inserted/deleted tuples in $\delta_{(x+1)x}$, and then join the (updated) relations corresponding to the nodes in G to derive $J_x(G)$. REQUERE optimizes this process by treating $J_x(G)$ as a view and applying the join view maintenance techniques [5] to derive the set of delta tuples $\delta J_{(x+1)x}$ such that $J_x(G) = J_{x+1}(G) \oplus \delta J_{(x+1)x}$. There are two benefits of this optimization technique: (1) avoiding the cost of updating the involved relations corresponding to nodes in G, and (2) exploiting the existing computation of the joins among the relations corresponding to G before modification (i.e., $J_{x+1}(G)$).

Example 4.4 *Continuing with Example 4.3,* REQUERE *derives $\mathcal{R}$ ={Master, Batting} as the set of core relations for T_2, and a schema subgraph G consisting of one edge Master-Batting to derive the IEQs. Since $J_2(G)$ has been computed in the previous step,* REQUERE *computes $J_1(G)$ from $J_2(G)$ using the view maintenance techniques. Finally,* REQUERE *uses* TALOS *to derive an IEQ $Q'_{1,1}$ w.r.t. $J_1(G)$: $\pi_{name,HR}\sigma_{HR>50}(Master \bowtie Batting)$. Thus,* REQUERE *returns $Q'_{1,1}$ as an IEQ of T_2 w.r.t the latest version D_1.* □

4.5 Multiple Database Versions & Multiple Known Results

This section addresses the second application scenario of deriving IEQs in the context of multiple database versions. The input to the problem consists of a sequence of database version and input table pairs, (D_1, T_1), (D_2, T_2), $\cdots$, (D_ℓ, T_ℓ); where T_i is produced by executing the same (unknown) query Q on the database version D_i. We assume that all databases D_i have the same schema, and that the query Q is monotonic (i.e., if $D \subseteq D'$, then $Q(D) \subseteq Q(D')$).

To simplify the presentation and without loss of generality, we discuss the solutions of REQUERE assuming $\ell = 2$; the techniques can be generalized to the case with $\ell > 2$. The key challenge here is to optimize the derivation of an IEQ Q' such that $Q'(D_i) = T_i$ for each $i \in [1, \ell]$. The intuition behind the optimization technique of REQUERE is to derive the following two data structures: (1) a "unified" database version $\mathcal{D}$ that is a combination of D_i's in some way, and (2) a "unified" input table $\mathcal{T}$ that is a combination of T_i's in some way such that the IEQs for $\mathcal{T}$ on $\mathcal{D}$ are also the queries that can reverse-engineer each T_i from D_i correspondingly. In the next discussions, we present the algorithms of REQUERE in two cases depending on whether the delta between D_2 and D_1 includes only the insertion operations or arbitrary operations.

4.5.1 Append-Only Database Versions

We first consider the simpler case where the database versions are "append-only" (i.e., $D_{i+1} \supseteq D_i$).

Straightforward Approach (REQUERE$^-$). To motivate the optimizations adopted by REQUERE to derive IEQs, we first present a simpler variant of REQUERE, denoted by REQUERE$^-$, which finds the IEQs on each $J_i(G)$ *separately*.

For each considered database version D_i, REQUERE$^-$ basically identifies sets of matching attributes S_i for T_i. Since REQUERE$^-$ needs to reverse-engineer the same query Q' for T_1 and T_2, each set of derived matching attributes for T_1 and T_2 must necessarily be the same. For each common set of matching attributes S, REQUERE$^-$ computes a set of core relations $\mathcal{R}$ corresponding to S and schema subgraphs G containing $\mathcal{R}$; followed by the join relation $J_i(G)$ by joining all the relations in G using foreign-key joins for D_i, $i \in \{1, 2\}$. REQUERE$^-$ will derive the IEQs of T_1 w.r.t. D_1, and the IEQs of T_2 w.r.t. D_2 *separately* until it can derive a common IEQ that is used to reverse-engineer T_i from D_i, $i \in \{1, 2\}$. The obvious drawback of this approach is that it might incur very high computational costs to generate IEQs before obtaining an IEQ that is derivable from both D_1 and D_2.

Example 4.5 *Consider two database versions: (i) D_1, the database example given in Figure 3-1, and (ii) D_2, given by the "delta" δ_{21} in Figure 4-4(a). Consider the following two given result tables $T_1 = \{(B, 73)\}$ and $T_2 = \{(B, 73), (F, 80)\}$, which are produced by executing the same (unknown) query Q on the database version D_1 and D_2 corre-*

	pID	name	year	stint	team	HR	Class label
τ_1	P1	A	2001	2	PIT	40	-
τ_2	P1	A	2003	2	ML1	50	-
τ_3	P2	B	2001	1	PIT	73	+
τ_4	P2	B	2002	1	PIT	40	-
τ_5	P3	C	2004	2	CHA	35	-
τ_6	P4	D	2001	3	PIT	30	-
τ_7	P5	E	2004	3	CHA	60	-
τ_8	P6	F	2004	3	SFN	80	+
τ_9	P2	B	2003	1	PIT	73	AL

$J_2(G) = Master \bowtie Batting$

Figure 4-5: Example of finding IEQs for multiple input tables

spondingly.

To derive the IEQ Q, REQUERE$^-$ *first finds the set of matching attributes S for both T_1 and T_2, and derives $S = \{Master.name, Batting.HR\}$. The set of core relations is $\mathcal{R} = \{Master, Batting\}$, and the considered subgraph G consists of one edge Master-Batting. The join relation $J_2(G)$ is shown in Figure 4-5, and the join relation $J_1(G) = \{\tau_i, i \in [1, 7]\}$.* REQUERE$^-$ *then derives an IEQ Q'_1 of T_1 w.r.t. D_1: $\sigma_{name,HR}\sigma_{HR>60}$ (Master $\bowtie$ Batting). Correspondingly,* REQUERE$^-$ *also derives Q'_1 as an IEQ of T_2 w.r.t. D_2. Thus,* REQUERE$^-$ *concludes that Q'_1 is an IEQ that can reverse-engineer T_i from D_i.* □

Optimization. In contrast to the simple approach of REQUERE$^-$ to find the IEQs for T_i w.r.t. $J_i(G)$ separately, our key optimization ideas for REQUERE are to find a "unified" relation $\mathcal{J}(G)$ that is a combination of $J_1(G)$ and $J_2(G)$, and a labeling scheme for tuples in $\mathcal{J}(G)$ such that the following two conditions are satisfied. First, the IEQs Q' derived from $\mathcal{J}(G)$ (if possible) are the IEQs of T_i w.r.t. D_i. Second, if there do not exist any IEQs on $\mathcal{J}(G)$, then there also do not exist any IEQs that can reverse-engineer T_i from D_i, $i \in \{1, 2\}$, at the same time.

Observe that $J_2(G) \supseteq J_1(G)$ and $T_2 \supseteq T_1$, since D_2 is derived from D_1 by combining with only inserted tuples. Therefore, REQUERE considers $J_2(G)$ as the unified relation in this case; i.e., $\mathcal{J}(G) = J_2(G)$.

Assume $T_2 = \{t_1, \cdots, t_k\}$, REQUERE partitions $J_2(G)$ into $(k + 1)$ disjoint partitions

$J_2(G) = J_0 \cup P_1 \cup \cdots \cup P_k$, where each P_i is the maximal subset of $J_2(G)$ that produces t_i when they are projected on $proj(Q)$; i.e., $\pi_{proj(Q)}(P_i) = t_i$. If we need to reverse-engineer only T_2 from D_2, then we need to label at least one tuple in each P_i as positive tuples for t_i to be present in the query result of the derived IEQ Q'. This semantics, referred to as *at-least-one semantics*, has been solved in TALOS. However, in our context, since we need to also reverse-engineer the same query Q' for T_1 w.r.t. D_1, the semantics is rather intricate. REQUERE proposes a new labeling scheme to solve this problem as follows. First, REQUERE labels tuples in J_0 as *negative* tuples, since they do not contribute to produce any tuples in T_2 (and T_1 as well). For the remaining tuples of $J_2(G)$, observe that $P_i \cap J_1(G)$ is the maximal subset of $J_1(G)$ that produces t_i when the tuples in this set are projected on their $proj(Q)$'s values. In the following, we describe how REQUERE labels the remaining tuples in $J_2(G)$ depending on whether $P_i \cap J_1(G)$ is empty or not.

1. If $P_i \cap J_1(G) = \emptyset$, it indicates that $t_i \in T_2 - T_1$. Therefore, REQUERE labels at least one tuple in P_i as positive for t_i to be present in $Q'(D_2)$.

2. If $P_i \cap J_1(G) \neq \emptyset$, REQUERE considers the following two cases:

   (a) If $t_i \in T_1$, then REQUERE labels at least one tuple in $P_i \cap J_1(G)$ as positive tuples for t_i to be present in $Q'(D_1)$. With this constraint, REQUERE also ensures that t_i will be present in $Q'(D_2)$ since $Q'(D_1) \subseteq Q'(D_2)$. The remaining tuples in $P_i - (P_i \cap J_1(G))$ are free to be labeled positive or negative tuples.

   (b) If $t_i \in T_2 - T_1$, then all tuples in $P_i \cap J_1(G)$ must be labeled negative. Otherwise, when some tuple in $P_i \cap J_1(G)$ is labeled positive, t_i will belong to $Q'(D_1)$; this fact contradicts to our assumption that $t_i \notin T_1$. For t_i to be present in $Q'(D_2)$, REQUERE labels at least one tuple in $P_i - (P_i \cap J_1(G))$ as positive tuples.

The core semantics in the two cases above are the *at-least-one* semantics, which REQUERE can utilize the existing techniques of TALOS to handle.

Example 4.6 *We discuss the optimization technique of* REQUERE *for Example 4.5. The unified relation in this case is $J_2(G)$.* REQUERE *partitions $J_2(G)$ into: $J_2(G) = J_0 \cup P_1 \cup P_2$, where $P_1 = \{\tau_3, \tau_9\}$ and $P_2 = \{\tau_8\}$. Since $P_1 \cap J_1(G) = \{\tau_3\}$ and $t_1 \in T_1$, using the reasoning in case (2a),* REQUERE *labels τ_3 as positive and allows τ_9 to be labeled positive or negative. Correspondingly, since $P_2 \cap J_1(G) = \emptyset$, using the reasoning in case (1),* REQUERE *labels τ_8 as positive tuple. Using this labeling scheme,* REQUERE *derives Q'_1: $\sigma_{name,HR}\sigma_{HR>60}$ (Master $\bowtie$ Batting) as an IEQ that can reverse-engineer T_i from D_i at the same time.* □

4.5.2 Arbitrary Database Versions

The ideas presented for the simpler case of append-only database versions can be extended to the general case of arbitrary database versions, where each "delta" version can consist of both inserted and deleted tuples. The unified join relation considered in this case is the combined join relation with inserted tuples only (without considering the deleted tuples). The way to label the tuples in this unified relation also follows the same scheme as described for the append-only case.

4.6 Supporting More Expressive IEQs

In this section, we present the techniques of REQUERE to support more expressive fragments of IEQs beyond the basic SPJ queries. The increased expressiveness is important to broaden the range of applications of QBO. To simplify the presentation, we assume the context of a single database version; the techniques can be easily extended to the general context with multiple database versions. Thus, in this section, the inputs to the QBO problem are a specific database D and an input table T, and the goal is to derive an IEQ Q' that belongs to a more expressive query fragment (i.e., Q' is a SPJU or SPJA query). The basic heuristic of REQUERE is to try to derive IEQs that belong to a simpler fragment before proceeding to the more complex fragments. Specifically, REQUERE derives IEQs using the following sequence of query fragments: SPJ, SPJU, and SPJA.

4.6.1 Finding SPJU-IEQs

REQUERE resorts to derive SPJU-IEQs when it fails to derive any SPJ-IEQs. Since an SPJU-IEQ Q' is a union of some n number of SPJ queries ($n > 1$), Q' is derived by partitioning $T = Q(D)$ into n subsets, $Q_1(D), \cdots, Q_n(D)$, where each of $Q_i(D)$ is produced by some SPJ-IEQ Q'_i. It is desirable to generate a succinct SPJU-IEQ Q' where n is minimized; however, this optimization problem is a hard problem (shown in Theorem 3.2). The heuristic adopted by REQUERE is to generate SPJ sub-queries Q'_i that form Q' iteratively in the non-increasing order of the number of tuples of T that $Q'_i(D)$ can contain.

Algorithm 4: REQUERE$_U(T, D, n, k)$

```
1  foreach column $C_i$ of $T$ do
2      $\mathcal{MA}_i \leftarrow \{(A, L)\}$ where $A$ is a schema attribute, $L = A \cap C_i$, and $|L| > 0$;
3  end
4  Enumerate sets $SL \subseteq \cup_{i=1}^{k} \mathcal{MA}_i$, where each $SL$ contains one element from each
   $\mathcal{MA}_i$;
5  $SMA = \emptyset$;
6  foreach $SL = \{(A_1, L_1, ), \cdots, (A_k, L_k)\}$ do
7      $SMA \leftarrow SMA \cup \{(A_1, \cdots, A_k, (L_1 \cap \cdots \cap L_k))\}$;
8  end
9  $i = 1$;
10 while $(i \leq n) \wedge |T| > 0$ do
11     Pick an element $(A_1, \cdots, A_k, L)$ from $SMA$ such that $|L \cap T|$ is largest among
       all possible elements from $SMA$;
12     $T' \leftarrow L \cap T$;
13     Use TALOS to derive an IEQ $Q'_i$ using the set of matching attributes
       $\{A_1, \cdots, A_k\}$ w.r.t. the input table $T'$;
14     $i \leftarrow i + 1$;
15     $T \leftarrow T - T'$;
16 end
17 return $Q'_1 \cup \cdots \cup Q'_i$;
```

The technique of REQUERE to derive SPJU-IEQs for T is sketched in Algorithm 4. Assume that there are k columns $C_1, \cdots, C_k$ in the given result table T. REQUERE first derives "partially" matching attributes for each column C_i of T, where a schema attribute A

partially matches C_i if the column of A contains some (not necessarily all) tuples of C_i (line 2). REQUERE then enumerates all sets of partially matching attributes, and stores in SMA (line 4-7). Each element of SMA is of the form $(A_1, \cdots, A_k, L)$; where A_i is a partially matching attribute of the corresponding C_i column, and L is the set of tuples of T that these attributes' columns together contain. In the next steps, REQUERE uses SMA to derive the SPJU-IEQs as follows.

Initially, all tuples in T are considered to be uncovered. REQUERE performs at most n iterations; at iteration i, REQUERE picks an element $(A_1, \cdots, A_k)$ from SMA that contains *the largest* number of uncovered tuples in T, denoted as T', and marks tuples in T' as covered (line 11). REQUERE then uses TALOS to generate an SPJ-IEQ Q'_i using the set of matching attributes $\{A_1, \cdots, A_k\}$ w.r.t. the input table T' (line 13). REQUERE repeats the loop until the number of iterations exceeds n or all the tuples of T become covered.

4.6.2 Finding SPJA-IEQs

REQUERE resorts to derive SPJA-IEQs when it fails to derive any IEQs in the simpler fragments (i.e., SPJ and SPJU).

Consider the simplest scenario when T consists of two columns C_g and C_a, where C_g is completely covered by a schema attribute $R_g.A_g$ and C_a is not covered by any schema attributes. REQUERE will generate an SPJA-IEQ Q' with the group-by operation on $R_g.A_g$ and an aggregation function (SUM, AVG, or COUNT) on some schema attribute A_a to account for the column C_a. The key challenge for REQUERE is to determine $R_a.A_a$ and the aggregation function.

The framework of REQUERE to derive SPJA-IEQs is described in Algorithm 5. To find candidate attributes for A_a, REQUERE enumerates different schema subgraphs G, each of which contains R_g. For each schema subgraph G that is being considered, REQUERE computes the join relation $J(G)$ by joining all the relations in G based on the foreign-key joins among relations corresponding to vertices in G (line 1-3). Each of the attributes in $J(G)$ (except for A_g) will be considered as a candidate for A_a. The key task is to

Algorithm 5: REQUERE$_A(T, D, R_g.A_g)$

```
1  Enumerate different schema subgraphs $G$, each of which contains $R_g$;
2  foreach schema subgraph $G$ do
3      Compute the join relation $J(G)$;
4      foreach attribute $A_x$ in $J(G)$, $A_x \neq A_g$ do
5          Label tuples of $J(G)$ when $A_x$ takes the role of $A_a$;
6          $found \leftarrow$ Find IEQs w.r.t. the labeled $J(G)$;
7          if $found = true$ then
8              return;
9          endif
10     end
11 end
```

determine whether an attribute A_x in $J(G)$ can take the role of A_a, and which aggregation function can be used with A_x (line 5).

Suppose that T contains n tuples in the form (g_i, a_i), where g_i is the domain value in C_g's column and a_i is the domain value in C_a's column. Conceptually, REQUERE partitions $J(G)$ into $(n+1)$ disjoint partitions: $J(G) = J_0 \cup P_1 \cup \cdots \cup P_n$, where each P_i, $i > 0$, is the maximum subset of tuples in $J(G)$ that have the projection on A_g as g_i; i.e., $P_i = \{t \in J(G) \mid t.A_g = g_i\}$.

Because the tuples in J_0 do not contribute to produce any tuples in T, REQUERE labels them as *negative* tuples. For each partition P_i, $i > 0$, REQUERE needs to label tuples in a subset $P'_i \subseteq P_i$ as *positive* tuples (with tuples in $P_i - P'_i$ are labeled as negative) such that P'_i can account for the corresponding tuple (g_i, a_i) in T. In other words, the aggregation function applying on A_x attribute values of tuples in P'_i will produce a_i. To derive a subset P'_i of P_i, REQUERE considers the following two cases:

- If a_i is an integer value and $0 < a_i < |P_i|$ for all $i \in [1, n]$, then REQUERE will first use COUNT as the aggregation function, and then use SUM or AVG as the aggregation function.

- If a_i is a real value or there exists some a_i such that $a_i > |P_i|$ or $a_i < 0$, then REQUERE uses SUM or AVG as the aggregation function.

SPJA-IEQs with COUNT function. In this case, REQUERE has the flexibility to assign positive class labels to exactly a_i tuples of P_i. This constraint can be formulated as the *exactly-k* semantics, where the technique to handle this semantics has been described in Section 3.2.

SPJA-IEQs with SUM/AVG function. Consider a partition P_i that has n tuples; assume that the values on A_x column of these tuples are $x_1, \cdots, x_n$. The problem of deriving a subset $P'_i \subseteq P_i$ can be formulated as selecting some values among $\{x_1, \cdots, x_n\}$ to put into P'_i s.t. the summation (or averaging) of values in P'_i is equal to a_i. This problem is formalized as the subset-sum/subset-average problem, stated as follows [11].

Problem 4.1 (Subset-Sum/ Subset-Average) *Given an array of n numbers $A = \{x_1, \cdots, x_n\}$ and a number K, the subset-sum problem (resp. subset-average problem) is to find an assignment for a set of n binary variables c_i (i.e., $c_i = 0$ or $c_i = 1$) such that: $\sum_{i=1}^{n} x_i \cdot c_i = K$ (resp. $\frac{\sum_{i=1}^{n} x_i \cdot c_i}{\sum_{i=1}^{n} c_i} = K$).* □

The subset-sum problem is a well-known NP-hard problem; REQUERE uses the standard pseudo-polynomial algorithm to solve the subset-sum problem in $O(Kn)$ and the subset-average problem in $O(Kn^2)$ [11].

After deriving each set P'_i from P_i, REQUERE labels the tuples in $J(G)$ correspondingly, and applies the remaining steps of TALOS to derive the IEQs on $J(G)$.

SPJA-IEQ Generalization. In the general case where $T = Q(D)$ contains more than two columns, the additional challenge for deriving an SPJA-IEQ Q' is to determine which of the columns in T are to be computed in Q' based on group-by operations. The heuristic adopted by REQUERE is to consider all the columns of T that are completely covered by some matching attributes to be used together with the group-by operator. Given a set of candidate group-by attributes S, each of the remaining columns in T will be derived in Q' by an aggregation function. The techniques to solve the basic scenarios when T contains two columns as described above can be extended to derive more complex SPJA-IEQs with multiple aggregation functions in this case.

	Original query	Size
T_1	$\pi_{S.name,N.name}\ \sigma_{S.acctbal>4000 \wedge N.regionkey<4}\ (supplier \bowtie nation)$	4383
T_2	$\pi_{C.name,N.name}\ \sigma_{C.acctbal>3000}\ (customer \bowtie nation)$	95264
T_3	$\pi_{P.name,S.name}\ \sigma_{PS.avaiqty>3000 \wedge S.acctbal>9500}\ (part \bowtie partsupp \bowtie supplier)$	24672
T_4	$\pi_{O.clerk,L.extendedprice}\ \sigma_{L.quantity<2 \wedge O.orderstatus=\text{“P”}}\ (lineitem \bowtie order)$	3719
T_5	$\pi_{C.name,N.name}\ \sigma_{mktsegment=\text{“BUILDING”} \wedge C.acctbal>100}\ (customer \bowtie nation)$ $\bigcup \pi_{S.name,N.name}\ \sigma_{S.acctbal>4000}\ (supplier \bowtie nation)$	32554
T_6	$\pi_{P.name,SUM(PS.supplycost)}\mathcal{G}_{P.name}\ \sigma_{PS.retailprice>2000}\ (part \bowtie partsupp)$	4950
T_7	$\pi_{C.name,SUM(O.totalprice)}\mathcal{G}_{C.name}\ \sigma_{C.acctbal>3000}\ (order \bowtie customer)$	63533

Table 4.2: Test queries

4.7 Experimental Study

In this section, we evaluate the effectiveness and efficiency of REQUERE for the generalized setting of the first variant of QBO with the three additional challenges: (1) the input query is unknown (Section 4.7.1), (2) there are multiple database versions (Section 4.7.2), and (3) the IEQs are in more expressive fragments (Section 4.7.3).

We used TPCH data set (with a database size 1GB) as the test database. Table 4.2 shows the seven test queries used in our experiments, where the third column shows the "query size" (i.e., the number of tuples in each test query's result $T_i(D)$). The four test queries T_1 to T_4 have been used in the experiments for TALOS in Section 3.6. The control knobs for the experiments in this section are set in the same way as described in the experiments for TALOS in Section 3.6.

We used MySQL Server 5.0.51 for our database system, and all algorithms were coded using C++ and compiled and optimized with GNU C++ compiler. Our experiments were conducted on a dual-core, 2.33GHz PC with 3.25GB RAM running Linux.

For each test query T_i, we first evaluated the query on the relevant data set D to compute its result $T_i(D)$, and then used this result as inputs for running REQUERE. Thus, our test queries in the experiments are really the result tables $T_i(D)$ and not the queries T_i, as the goal of this study is to reverse-engineer the queries for the given result tables. The timings reported in this experiment are the time to derive *the first precise* IEQs for the given result tables. The comparisons between REQUERE and the corresponding non-

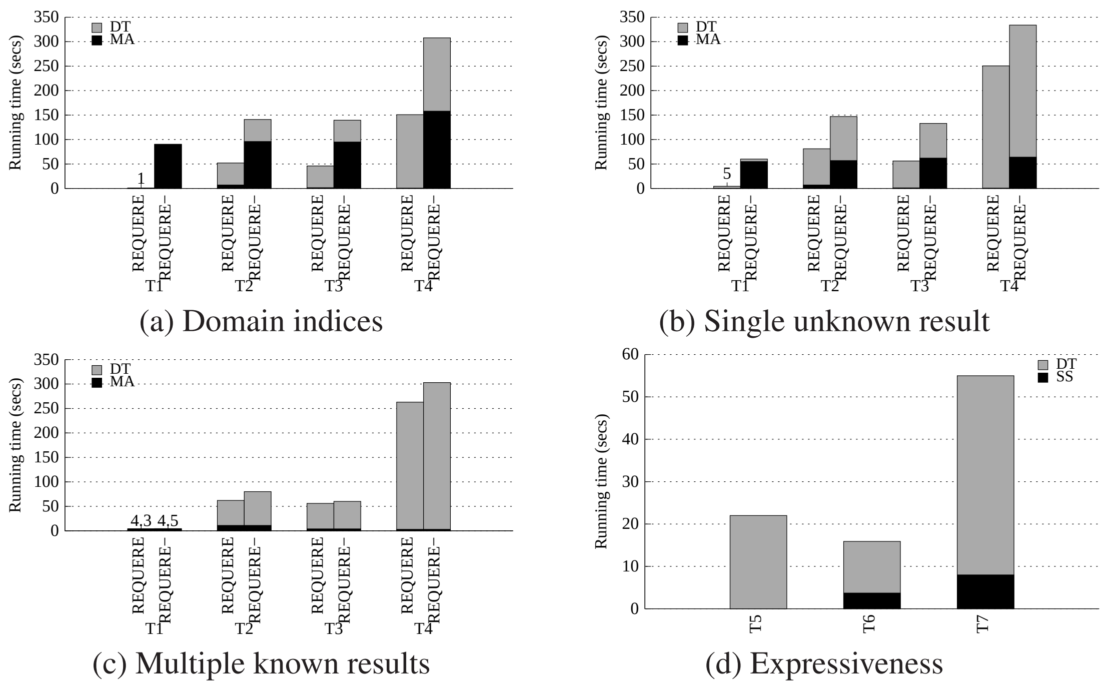


Figure 4-6: Experimental Results of REQUERE

optimized approach, `REQUERE`$^-$, are on the running time only; the number and quality of IEQs produced by `REQUERE` and `REQUERE`$^-$ are the same, as these qualities are independent of the optimizations.

In the first two sets of experiments (Sections 4.7.1 and 4.7.2), we analyze the running time to derive the IEQs for a given result table T, which consists of two components: (1) MA, the time to find the matching attributes for columns in T, and (2) DT, the time to derive an IEQ after the matching attributes have been determined.

4.7.1 Effectiveness of Domain Indices

In this section, we consider the setting of `QBO` when the inputs include a single result table and a single database version. We used four given result tables $T_i(D)$, $i \in [1,4]$, which are the results of evaluating T_i on the current version of TPCH data set. This experiment evaluates the efficiency of using domain indices to find matching attributes by comparing `REQUERE` with its variant, `REQUERE`$^-$, which does not make use of domain indices. The running times of `REQUERE` and `REQUERE`$^-$ differ from each other for the first component to derive matching attributes (i.e., MA), and are the same for the second component (i.e., DT). The result in Figure 4-6(a) shows that REQUERE runs much more efficiently than REQUERE$^-$ in finding matching attributes (i.e., MA component); i.e. REQUERE performs this step in the order of less than 10 seconds versus 90 seconds by REQUERE$^-$. Totally, REQUERE runs in the order of 1.5 and up to two-magnitude times faster than REQUERE$^-$. For these queries, REQUERE can successfully reverse-engineer the original queries T_i for the four given result tables $T_i(D)$.

Storage Overhead. The storage overhead of REQUERE consists of pre-computed domain index tables. In our experiment, the domain indices built for TPCH data set consist of 272MB over 1GB size of the database.

4.7.2 Multiple Database Versions

In this section, we study the effectiveness of REQUERE to derive IEQs when there are multiple database versions. We created one database delta including the modifications on *supplier*, *customer*, *partsupp*, *order* relations (these relations are used in the test queries T_1 to T_4). The modification in each relation R consists of 10% (of the number of tuples in R) randomly inserted tuples. Thus, there are two database versions, the current database version of TPC-H is the second version. We run the experiments for REQUERE in the following two scenarios.

Single Unknown Result. We study the efficiency of REQUERE to derive IEQs for a single unknown result table w.r.t. multiple database versions. For this task, we created the given result table $T_i(D)$ as the result of evaluating T_i on the first database version. Note that if we set $T_i(D)$ as the result of evaluating T_i on the current (i.e., the second) database version, both REQUERE and REQUERE$^-$ run in the same time, since both methods could reverse-engineer the queries w.r.t. the current database version in these cases.

Figure 4-6(b) shows that REQUERE$^-$ runs slower than REQUERE in the magnitude of 1.5 times. The reason is that REQUERE$^-$ spends a lot of time for the first step of finding matching attributes (i.e., MA component). In contrast, with the optimization strategy, REQUERE saves a lot of computation; i.e., MA is in the order of 10 seconds in REQUERE versus 60 seconds in REQUERE$^-$.

Multiple Known Results. In this set of experiments, we evaluate T_i on two database versions D_2 and D_1 to obtain two given result tables $T_i^2(D)$ and $T_i^1(D)$. Thus, the inputs to QBO include two pairs: $(D_2, T_i^2(D))$ and $(D_1, T_i^1(D))$.

We compare the optimization technique of REQUERE to derive the IEQs given a sequence of database version and input table (D_i, T_i) against a non-optimized approach (REQUERE$^-$) that finds the IEQs on each pair (D_i, T_i) separately. Figure 4-6(c) shows the effectiveness of REQUERE outperforming REQUERE$^-$ by a factor of 1.4 times faster. Among the two components of the running time to derive the IEQs, both REQUERE and REQUERE$^-$ spend the same time to derive matching attributes (i.e., MA); their difference lies in the time to derive the IEQs (i.e., DT). While REQUERE only needs to derive the IEQs on one "unified" hub table, REQUERE$^-$ needs to derive the IEQs on two separate hub tables. Therefore, REQUERE runs more efficiently than REQUERE$^-$.

For all the cases studied in this section, REQUERE can successfully reverse-engineer the original query T_i for the given result table $T_i(D)$ w.r.t. the correct database version.

4.7.3 Supporting More Expressive IEQs

In this section, we evaluate the effectiveness of REQUERE to derive IEQs in more expressive fragments (i.e., SPJ + union/aggregation operator). We analyze the running time of REQUERE in this section including two components: (1) SS, the time to derive the candidate attributes for the aggregated column (to derive SPJA-IEQs), and (2) DT, the other steps to derive the IEQs. Note that the first component does not appear for deriving SPJU-IEQs.

Query T_5 is an example of SPJU query; REQUERE was able to reverse-engineer the query from the corresponding result table $T_5(D)$ in 22 seconds.

Queries T_6 and T_7 are examples of SPJA queries. REQUERE was also successful to reverse-engineer the original queries for the given result tables $T_6(D)$ and $T_7(D)$. The time to derive the candidate attributes for aggregated column (by solving the derived subset-sum/subset-average problem) consists of 20% the total running time to derive the IEQs.

4.8 Summary

In this chapter, we have generalized the first setting of `QBO` to derive instance-equivalent queries with the following three additional challenges: (1) the original query is not given as part of the input, (2) the derived queries are more expressive and go beyond the simple Select-Project-Join query fragment, and (3) there are multiple database versions. We presented a generalized approach (`REQUERE`) to address these issues, and demonstrated its effectiveness and efficiency with an experimental evaluation on real data sets.

Chapter 5

ConQueR: Explaining Why-Not Questions

One useful feature that is missing from today's database systems is an explain capability that enables users to seek clarifications on unexpected query results. There are two types of unexpected query results that are of interest: the presence of unexpected tuples, and the absence of expected tuples (i.e., missing tuples). Clearly, it would be very helpful to users if they could pose follow-up *why* and *why-not* questions to seek clarifications on, respectively, unexpected and expected (but missing) tuples in query results. While the why questions can be addressed by applying established data provenance techniques, the problem of explaining the why-not questions has received very little attention.

In this chapter, we introduce our new paradigm for explaining a why-not question, highlighted in Table 5.1, that is based on automatically generating a refined query, whose result includes both the original query's result as well as the user-specified missing tuples. We present an overview of our explanation model and a novel framework, named `ConQueR`, to explain why-not questions in Section 5.1, followed by the techniques of `ConQueR` in Sections 5.2 - 5.4. We then discuss an alternative solution `TALOS`$^+$, which is adapted from `TALOS` designed for the first variant of `QBO` problem, to provide explanations for why-not questions in Section 5.5. We also discuss how to use `TALOS`$^+$ and

ConQueR for deriving IEQs and explaining why-not questions in Section 5.5. We describe the implementation details of ConQueR in Section 5.6. To conclude this part of work, we present the experiment studies in Section 5.7 to compare the performance of ConQueR against TALOS$^+$ in terms of the processing efficiency and the quality of the derived refined queries. We also compare the effectiveness of our query-refinement based approach of explaining why-not questions against the two existing approaches in [9, 22]. Finally, we summarize our work on explaining why-not questions in Section 5.8. Part of the contents and materials in this chapter were previously published in [48].

QBO Problem	Parameters	
	Input query Q	Given result table T
The first variant	- Q is known - Q is unknown	$T = Q(D)$ T is a set of specific tuples
The second variant	**Q is known**	**T = Q(D) $\bigcup$ S** **S is a set of tuples that are not present in Q(D)**
The third variant	Q is partially specified	T is a set of constraints that must be satisfied by the query result of each derived query Q'

Table 5.1: The Focus of Chapter 5

5.1 An Overview of ConQueR

In this work, we consider three fragments of SQL queries in ConQueR, where each projected attribute is either a relation's attribute or a value computed by an aggregation operator (COUNT, SUM, or AVG) that does not involve any arithmetic expression in the operator's argument. The first fragment is the basic Select-Project-Join (SPJ) queries, where the selection condition is a conjunction of predicates $C_1 \wedge \cdots \wedge C_\ell$. Each C_i is either a selection predicate "A_j *op* c" or a join predicate "A_j *op* A_k", where A_i is an attribute, c is a constant, and *op* is a comparison operator. The second fragment is SPJ queries with aggregation (SPJA queries), which are SPJ SQL queries with aggregation operators in the select-clause and an optional group-by clause. The third fragment is SPJ queries with union operator (SPJU) of the form Q_1 *union* $Q_2 \cdots$ *union* Q_n, where

pID	*name*
P1	“A”
P2	“B”
P3	“C”
P4	“D”
P5	“E”

(a) *Player*

pID	**team**	**year**	*pts*	*blk*	*stl*	*reb*
P1	GSW	1973	2009	30	150	40
P2	SEA	1994	1689	35	200	281
P2	SEA	1995	1563	50	240	339
P3	CHI	1992	2541	45	220	361
P4	LAL	1995	1567	30	162	359

(b) *Regular*

pID	**team**	**year**	*pts*	*blk*	*stl*	*reb*
P1	GSW	1973	2029	40	100	30
P2	SEA	1994	3000	65	150	181
P2	CHI	1995	2200	50	120	161
P2	LAL	1996	2500	70	110	200
P4	LAL	1995	2300	70	150	150
P5	DEN	2000	1689	35	200	381

(c) *Playoff*

Figure 5-1: Running Example: Basketball Data Set D

each Q_i is an SPJ query.

For simplicity and without loss of generality, we assume all attributes are numerical and consider only the “$\leq$” comparison operator for selection predicates. Our approach can be easily extended to other comparison operators. In Section 5.4, we will discuss how ConQueR handles categorical attributes in the query’s selection conditions.

Based on the knowledge of the primary and foreign key constraints in the database, the database schema can be modeled as a *schema graph*, denoted by $\mathcal{SG}$. Each node in $\mathcal{SG}$ represents a relation, and each edge between two nodes represents a foreign-key join between the relations corresponding to the nodes.

Running example. We use the NBA statistics database on basketball players as our running example. The *Player* relation contains the identifier (*pID*) and name (*name*) of each player. The *Regular* (resp. *Playoff*) relation provides the number of points (*pts*), block (*blk*), steal (*stl*), and rebound (*reb*) statistics of a player when he was playing for a team (*team*) in a specific year (*year*) in regular season (resp. playoff) games. Figure 5-1 shows our running example data.

5.1.1 Why-not Questions & Refined Queries

Given an input query Q on a database D, let $Q(D)$ denote the output of Q on D. In the most basic form, a *why-not question* on $Q(D)$ is represented by a non-empty set of *why-not tuples* $S = \{t_1, \cdots, t_n\}$, $n \geq 1$, where each why-not tuple t_i has the same schema as Q and $t_i \notin Q(D)$. Essentially, the why-not question is asking why S is not a subset of $Q(D)$; i.e., why each $t_i \in S$ is not in $Q(D)$. Each component value in a why-not tuple can be in one of three forms: (1) a constant value compatible with the corresponding attribute's domain; (2) a don't-care value (denoted by _); or (3) a variable (denoted by a \$ symbol followed by a sequence of letters; e.g., \$$x$). A don't-care value is used for an attribute A_i when the user is not interested in the specific value of attribute A_i in the why-not tuple. A variable is used for an attribute A_i when the user wishes to impose a selection condition on that attribute in the why-not tuple with respect to some constant value or another attribute appearing in the same or another why-not tuple, as illustrated by the SPJA queries in Examples 1.4 and 1.5. Thus, in the most general form, a why-not question on $Q(D)$ is represented by (S, C); where S is a non-empty set of why-not tuples, and C is a (possibly empty) set of selection conditions defined on the variables appearing in S. In Example 1.5, the why-not question is represented by $S = \{$(Alice,\$$x$), (Bob,\$$y$)$\}$ and $C = \{\$x > \$y\}$.

Given a why-not question (S, C) on $Q(D)$, we say that Q' is a *refined query* of Q that explains the why-not question (S, C) if (1) $Q'(D)$ contains[1] $Q(D)$, and (2) for each why-not tuple $t_w \in S$, there exists a *matching tuple* $t \in Q'(D)$ such that all the constraints in C are satisfied by the matching tuples. A tuple $t \in Q'(D)$ is a *matching tuple* for a why-not tuple $t_w \in S$ if for every component of t_w that is a constant value, the corresponding component in t has the same constant value. Thus, if t is a matching tuple for t_w, then every component of t_w that is a variable becomes instantiated with the corresponding attribute value in t, and the collection of instantiated variables must satisfy all the constraints in C

[1]For certain cases where C involves a constraint specification, the attribute values associated with the constraints could be different between $Q'(D)$ and $Q(D)$.

for Q' to be a refined query of Q w.r.t. D.

5.1.2 Metrics for Comparing Refined Queries

Since there are generally many refined queries for a given why-not question, it is useful to have some metric to compare the quality of refined queries so that only the "good" refined queries are returned as possible explanations. There are two obvious desiderata for refined queries that can be used for this purpose.

Dissimilarity metric. First, a refined query should be as similar as possible to the original input query. This has intuitive appeal since a refined query that is minimally modified from the original query is likely to retain as much of the intention of the original input query. Moreover, by comparing the small differences between the two queries, it also serves to pinpoint to the user the "errors" she has made in her initial query. Thus, a refined query that simply modifies only one of the selection predicate is more similar to the input query than another refined query that involves a different set of relations from the original query.

Given an input query Q and a refined query Q', we compare the similarity of Q and Q' by measuring the minimum edit distance of transforming Q to Q'. Thus, two queries are more similar (or less dissimilar) if their edit distance is smaller. Since the output of Q and Q' are union-compatible (i.e., the lists of attributes in the select-clause of Q and Q' are the same), we only consider edit operators to transform Q to Q' in terms of modifying the query's from-clause and where-clause. The corresponding modifications to the query's select-clause and group-by-clause are trivial and not considered in the edit distance computation. The four key edit operations considered are: (O_1) modify the constant value of a selection predicate in the where-clause, (O_2) add a selection predicate in the where-clause, (O_3) add/remove a join predicate in the where-clause, and (O_4) add/remove a relation in the from-clause. Note that there is no explicit edit operator for removing a selection predicate as this can be modeled by O_1; i.e., the removal of a selection predicate is effectively equivalent to modifying its range of selection values to cover the whole domain of the attribute. Furthermore, when O_4 is used to remove a relation R_i in the from-clause, all the selection and join predicates that are associated with R_i are also removed as a part of the edit operation.

Let w_i denote the cost of the edit operation O_i, $i \in [1, 4]$. It is reasonable to assume that $w_1 < w_2 < w_3 < w_4$[2]. Let n_i denote the total number of O_i operations used in a transformation of Q to Q', $i \in [1, 4]$. The edit distance for this transformation is given by $\sum_{1 \leq i \leq 4}(w_i \times n_i)$. We refer to the minimum edit distance to transform Q to Q' as the *dissimilarity measure* between Q and Q', which can be computed efficiently for a given pair Q and Q'.

Imprecision metric. Second, the refined query should be as precise as possible in terms of its result. Ideally, the result of the refined query Q' should contain only the result of the original query Q and the set of matching tuples for the why-not tuples. Any additional tuples returned in $Q'(D)$ are considered to be irrelevant tuples that should be minimized. Given a refined query Q' for a why-not question (S, C), let $R \subseteq Q'(D)$ denote a minimal set of matching tuples in $Q'(D)$ for the why-not tuples in S. The imprecision metric for Q' is defined to be the number of irrelevant tuples in $Q'(D)$, which is given by $|Q'(D) - Q(D) - R|$.

Skyline refined queries. Thus, a refined query is considered to be good if both its dissimilarity and imprecision metrics are low. Among all the possible refined queries for a why-not question, we are interested in the set of skyline refined queries defined as follows [6]. Given two different refined queries Q_1 and Q_2, we say that Q_1 *dominates* Q_2 if (1) both the metrics of Q_1 are at least as low as those of Q_2, and (2) for at least one of the metrics, Q_1's value is strictly lower than that of Q_2's. We define a refined query Q' to be a *skyline refined query* (or skyline query) if Q' is not dominated by any other refined query. Thus, given a why-not question, our goal is to compute skyline refined queries to explain the question.

[2]In our experimental study, we use $w_1 = 1$, $w_2 = 3$, $w_3 = 5$ and $w_4 = 7$.

Example 5.1 *Consider a query Q_1 on the Basketball data set that finds players who have "block" statistics no greater than* 30 *and "steal" statistics no greater than* 150*; i.e., Q_1: SELECT name FROM Player P, Regular R WHERE P.pID = R.pID AND blk* $\leq$ 30 *AND stl* $\leq$ 150*. The output includes only one player "A". The why-not question $S = \{(\text{"B"})\}$ asks why player "B" is excluded from the result.*

Consider the following refined query Q'_1: SELECT name FROM Player P, Regular R WHERE P.pID = R.pID AND blk $\leq$ 35 *AND stl* $\leq$ 200*. Observe that Q'_1 is derived from Q_1 by applying O_1 edit operation on both the selection predicates of Q_1, and the output of Q'_1 is {"A", "B", "D"}. Thus, the dissimilarity and the imprecision of Q'_1 (w.r.t. Q_1) are* $2w_1$ *and* 1*, respectively.*

Consider yet another refined query Q''_1: SELECT name FROM Player P, Regular R WHERE P.pID = R.pID AND blk $\leq$ 50 *AND stl* $\leq$ 240*. The output of Q''_1 is {"A", "B", "C", "D"}; and the dissimilarity and imprecision of Q''_1 are* $2w_1$ *and* 2*, respectively. Thus, Q'_1 dominates Q''_1, and Q'_1 is considered to be a better refined query than Q''_1.* □

5.1.3 Explaining with ConQueR

In this section, we present an overview of our approach named ConQueR, for **Con**straint-based **Que**ry **R**efinement, to explain why-not questions by automatically generating one or more refined queries.

Indeed, we have extended TALOS, the classification-based approach designed for the first variant of QBO, as an alternative solution for explaining why-not questions (the details are provided in Section 5.5). Since TALOS is a more precision-oriented approach, the queries generated can be rather different from the input query. In some applications, it may not be too meaningful to explain missing tuples using refined queries that are very different from the input query. Moreover, the performance of TALOS is also slower than ConQueR due to its costly data classification step. A more detailed comparison between ConQueR and TALOS will be given in Section 5.5.4.

ConQueR is designed to be a *similarity-driven* approach in that it tries to generate refined queries with low dissimilarity values before considering more precise refined queries that have higher dissimilarity values. Given a why-not question (S, C) for a query Q on database D, ConQueR will first consider refined queries Q' that have the same query schema (i.e., queries with the same from-clause and join predicates) as Q. That is, ConQueR tries to derive Q' by simply modifying selection predicate(s) in Q to explain the why-not tuples while minimizing the imprecision metric. If such refined queries exist, ConQueR will only generate skyline refined queries that all share the same query schema as Q. However, if no such refined query exists, ConQueR then looks for refined queries that have a slightly different query schema (i.e., with a slightly higher dissimilarity value), and so on. Thus, ConQueR effectively iterates over a sequence of query schemas $QS_1, \cdots, QS_k$ to search for refined queries: QS_1 is the query schema of the input query Q, and schema QS_{i+1} is considered only if there are no refined queries with schema $QS_1, \cdots, QS_i$. The sequence of query schemas considered are (approximately) of increasing dissimilarity metric values, and if QS_k is the first query schema in the sequence to contain refined queries, ConQueR will generate all skyline refined queries with schema QS_k as possible explanations to the why-not question.

The architecture of ConQueR consists of two key components, ConQueR^s and ConQueR^p. Given a query Q on a database D and a why-not question (S, C) on $Q(D)$, ConQueR^s will first compute a refined query Q'_s for the why-not question such that Q'_s is as similar as possible to Q (i.e., Q'_s has a low dissimilarity value). Next, ConQueR^p will use Q'_s to derive skyline refined queries Q'_p that are more precise than Q'_s. Specifically, Q'_p is derived from Q'_s by adding various additional predicates to Q'_s to improve its precision.

Notations & Definitions. Given a SQL query Q, we use $rel(Q)$ to denote the set of relations in the from-clause of Q; $proj(Q)$ to denote the set of attributes in the select-clause of Q; $sel(Q)$ to denote the set of selection predicates in the where-clause of Q; and $join(Q)$ to denote the set of join predicates in the where-clause of Q. Thus, the *query schema* of a query Q is given by $rel(Q)$ and $join(Q)$. We use ℓ to denote the number of selection predicates in Q; i.e., $|sel(Q)| = \ell$.

	pID	name	team	year	*pts*	*blk*	*stl*	*reb*
t_1	P1	A	GSW	1973	2009	30	150	40
t_2	P2	B	SEA	1994	1689	35	200	281
t_3	P2	B	SEA	1995	1563	50	240	339
t_4	P3	C	CHI	1992	2541	45	220	361
t_5	P4	D	LAL	1995	1567	30	162	359

$Q^*_\emptyset(D) = Player \bowtie_{pID} Regular$

Figure 5-2: Example 5.2

Consider the generation of a refined query Q' for a why-not question (S, C) on $Q(D)$ that shares the schema as Q. Conceptually, ConQueR first computes an intermediate query, denoted by $Q^*_\emptyset$, on D, where $rel(Q^*_\emptyset) = rel(Q)$, $join(Q^*_\emptyset) = join(Q)$, $sel(Q^*_\emptyset) = \emptyset$, and $proj(Q^*_\emptyset)$ consists of all the distinct attributes in $rel(Q^*_\emptyset)$. The refined query Q' is derived from $Q^*_\emptyset(D)$ as follows: $Q' = \pi_L(\sigma_P(Q^*_\emptyset))$, where $L \subseteq proj(Q^*_\emptyset)$ is a list of appropriate attributes corresponding to $proj(Q)$ so that Q and Q' are union-compatible, and P contains an appropriate set of selection predicates such that Q' is a refined query for the why-not question. Determining L from Q and $proj(Q^*_\emptyset)$ is straightforward, and the main challenge in the derivation of Q' is determining P (i.e., $sel(Q')$).

For each why-not tuple $t_i \in S$, let $M_i \subseteq Q^*_\emptyset(D)$ denote the subset of tuples in $Q^*_\emptyset(D)$ that are the matching tuples of t_i. Note that for Q' to be a refined query of Q that explains all the why-not tuples, it is necessary for each M_i to be non-empty; otherwise, if some M_j is empty, then Q' will not be able to account for the why-not tuple t_j.

Example 5.2 *Consider again query Q_1 in Example 5.1, where $Q_1(D) = \{(\text{“A”})\}$. Consider the derivation of a refined query Q' (with the same schema as Q_1) to explain a why-not tuple t_w =(“B”). The intermediate query $Q^*_\emptyset$ to derive Q' has $rel(Q^*_\emptyset) = \{Player, Regular\}$ and $join(Q^*_\emptyset) = \{Player.pID = Regular.pID\}$. The output of $Q^*_\emptyset$ on D is shown in Figure 5-2, and the set of matching tuples in $Q^*_\emptyset(D)$ for t_w is given by $M_w = \{t_2, t_3\}$. Thus, $Q'(D)$ needs to include t_2 or t_3 in order to account for the why-not tuple t_w.* □

5.2 Explaining SPJ Queries

This section presents how ConQueR generates refined queries Q' to explain why-not questions (S, C) on SPJ queries Q. We first consider the simpler case where Q' and Q share the same schema: Section 5.2.1 explains how ConQueR^s generates refined queries with low dissimilarity values, and Section 5.2.2 explains how ConQueR^p enhances these queries to improve their precision. Section 5.2.3 considers the more general case where the schema of Q and Q' are different.

For simplicity, we assume that there are no variables in the why-not tuples, and therefore also no constraints on the why-not tuples (i.e., $C = \emptyset$). Details on how ConQueR handles general SPJ queries are given in Section 5.4.

5.2.1 Modifying Selection Predicates

In this section, we explain how ConQueR^s generates a refined query Q' that has the same schema as Q. To maximize the similarity of Q' and Q, ConQueR^s derives Q' from Q by simply modifying some selection predicate(s) in Q. To simplify the presentation, we first consider the scenario where there is exactly one why-not tuple (i.e., $S = \{t_1\}$), and discuss the handling of the general scenario with multiple why-not tuples at the end of this section.

For simplicity and without loss of generality, let the selection predicates in Q be of the form: $sel(Q) = \{A_1 \leq v_1, \cdots, A_\ell \leq v_\ell\}$, $\ell \geq 1$. Since Q' is derived from Q by modifying some selection predicates, let v'_i denote the modified value of v_i in Q', for $i \in [1, \ell]$.

Let Q^* denote the query that is exactly the same as Q except that $proj(Q^*)$ includes all the distinct attributes in $rel(Q)$; i.e., $Q^* = \sigma_P(Q^*_\emptyset)$ where $P = sel(Q)$. Thus, $Q^*(D)$ is the subset of tuples in $Q^*_\emptyset(D)$ that form $Q(D)$ when $Q^*(D)$ is projected on $proj(Q)$. For each selection predicate attribute A_i, $i \in [1, \ell]$, define $v_i^{max} = max_{t \in Q^*(D)}(t.A_i)$. For $Q'(D) \supseteq Q(D)$, we must have $v'_i \geq v_i^{max}$, for $i \in [1, \ell]$.

For Q' to account for the why-not tuple t_1, $Q'(D)$ must contain at least one tuple from M_1[3]. However, to minimize the imprecision of Q', $Q'(D)$ should not contain more than one tuple from M_1. Thus, each tuple in M_1 contributes to a refined query Q'. For $Q'(D)$ to contain a tuple $t_m \in M_1$, we must have $v'_i \geq t_m.A_i$, for $i \in [1, \ell]$. Therefore, combining the two requirements above, for $Q'(D)$ to contain t_m and $Q'(D) \supseteq Q(D)$, $sel(Q')$ is specified by setting $v'_i = \max\{v_i^{max}, t_m.A_i\}$, for $i \in [1, \ell]$. Note that while it is possible to generate other refined queries Q'' that also satisfy the two requirements by setting some $v'_j > \max\{v_j^{max}, t_m.A_j\}$, the imprecision of Q'' will be at least as high as that of Q'; it means that Q'' will be dominated by Q'. Therefore, to generate only skyline refined queries, we must have $v'_i = \max\{v_i^{max}, t_m.A_i\}$, for $i \in [1, \ell]$.

In addition, since we are interested only in skyline refined queries, the number of refined queries considered can be reduced by considering only the "skyline" tuples in M_1. Consider two tuples $t_x, t_y \in M_1$, and let Q'_x and Q'_y denote the refined queries corresponding to t_x and t_y, respectively. We say that t_x dominates t_y if (1) $t_x.A_i \leq t_y.A_i$ for $i \in [1, \ell]$, and (2) at least one of the inequalities in (1) is strict. The skyline tuples in M_1 are the tuples that are not dominated by any tuple in M_1. If t_x dominates t_y, it follows that Q'_x dominates Q'_y. Thus, to generate skyline refined queries, we only need to consider the skyline tuples in M_1.

Example 5.3 *Reconsider Example 5.1, where the input query is Q_1 and the why-not tuple is t =("B"). Let A_1 and A_2 denote the two selection attributes blk and stl, respectively. We have $Q^* = \sigma_{blk \leq 30 \wedge stl \leq 150}(Q^*_\emptyset)$. Thus, $Q^*(D) = \{t_1\}$, $v_1^{max} = 30$, and $v_2^{max} = 150$. Since $M_1 = \{t_2, t_3\}$, there are two possible refined queries corresponding to these matching tuples for t. To generate the refined query Q'_1 such that $Q'_1(D)$ contains $t_2 \in M_1$,* ConQueR^s *modifies the two predicates in $sel(Q)$ into $blk \leq 35$ and $stl \leq 200$, and obtains the refined query Q'_1 as shown in Example 5.1.*

Similarly, to generate the refined query Q''_1 such that $Q''_1(D)$ contains $t_3 \in M_1$,

[3]Note that since $proj(Q') \subseteq proj(Q^*_\emptyset)$ and $M_i \subseteq Q^*_\emptyset(D)$, when we say that $Q'(D)$ must "contain" one tuple t from M_i, what we mean is that $Q'(D)$ must contain one tuple t that is a projection of some tuple t_{int} from M_i; i.e., $t = \pi_L(t_{int})$, where $L = proj(Q')$.

ConQueR^s *modifies the two predicates in sel(Q) into blk* ≤ 50 *and stl* ≤ 240*, and obtains the refined query* Q_1'' *as given in Example 5.1.*

However, by considering only the skyline tuples in M_1*,* ConQueR^s *actually would not have considered* Q_1'' *since* t_3 *is dominated by* t_2*, which means that* Q_1'' *is not a skyline refined query.* □

Finally, to generate the skyline refined queries from the set of queries corresponding to the skyline tuples in M_1, ConQueR^s needs to compute and compare the imprecision values of these queries by computing their results.

Handling multiple why-not tuples. The above technique can be easily extended to handle the general case where there are multiple why-not tuples; i.e., $S = \{t_1, \cdots, t_n\}$, $n > 1$. Specifically, for each M_i, $i \in [1, n]$, ConQueR^s first computes the set of skyline tuples, denoted by SL_i, in M_i. Next, ConQueR^s enumerates different refined queries Q' that correspond to different subsets $M' \subseteq \cup_{i=1}^{n} SL_i$ of matching tuples, where each M' consists of one tuple from each of SL_i, $i \in [1, n]$. For example, if t_j' is the tuple selected from each SL_j, $j \in [1, n]$, then the selection condition in the refined query Q' is specified by setting $v_i' = \max\{v_i^{max}, t_1'.A_i, \cdots, t_n'.A_i\}$, $i \in [1, \ell]$.

5.2.2 Improving Precision with More Predicates

Since the refined queries Q' produced by ConQueR^s are generated by simply modifying the selection predicates in Q, there are likely to be many irrelevant tuples in $Q'(D)$. In this section, we explain how ConQueR^p improves the precision of the refined queries Q' produced by ConQueR^s by adding additional selection predicates to Q' to reduce the irrelevant tuples in $Q'(D)$, while ensuring that the enhanced query Q' remains a refined query for the input why-not question. Thus, the refined queries produced by ConQueR^p tradeoffs low dissimilarity values for low imprecision values.

Consider a refined query Q' produced by ConQueR^s that corresponds to the subset of matching tuples $T \subseteq \bigcup_{i \in [1,n]} M_i$ to explain the set of why-not tuples $S = \{t_1, \cdots, t_n\}$.

Let $\mathcal{A}$ denote the set of attributes in $rel(Q')$ that do not have a selection predicate in $sel(Q')$. For each attribute $A_i \in \mathcal{A}$, ConQueR^p can add the following predicate[4] to try to reduce the irreverent tuples in $Q'(D)$: "$A_i \leq max_{t \in Q^*(D) \cup T}(t.A_i)$".

Thus, there are a total of $|\mathcal{A}|$ possible additional predicates that ConQueR^p can introduce into Q' to reduce its imprecision. As the problem to maximize the elimination of irrelevant tuples using the minimum number of additional predicates is NP-hard (shown by reduction from the Set-Covering problem[5]), ConQueR^p uses a standard greedy heuristic to select the additional selection predicates. In particular, ConQueR^p chooses the predicates in non-increasing order of the number of irrelevant tuples that they can eliminate.

5.2.3 Refined Queries with Different Schema

When ConQueR is unable to find refined queries having the same query schema as Q, ConQueR will consider other similar schemas, roughly in increasing order of their dissimilarity metrics. In this section, we explain how ConQueR enumerates alternative query schemas and generates refined queries for such schemas.

Enumerating schemas. ConQueR uses a simple heuristic to enumerate query schemas approximately in increasing order of dissimilarity metrics. Let S_R denote the set of the relations in $rel(Q)$ that contain the attributes in $proj(Q)$. ConQueR retains these relations in Q' so that the $proj(Q')$ and $proj(Q)$ are equal and Q' is more similar to Q[6]. Thus, ConQueR generates a different schema that contains all relations in S_R. The approach of ConQueR is similar to that of TALOS, which performs a bread-first-search traversal of the schema graph $\mathcal{SG}$ starting from an arbitrary vertex R_s in S_R. ConQueR keeps a queue $\mathcal{QG}$ of "active" schema subgraphs; $\mathcal{QG}$ is initialized with one schema subgraph G_s containing the vertex R_s. In each round, ConQueR picks from $\mathcal{QG}$ an active schema subgraph G, and

[4]In practice, ConQueR also considers to add the following predicate: "$A_i \geq min_{t \in Q^*(D) \cup T}(t.A_i)$".

[5]The proof of this result is similar to the proof of Theorem 6.2.

[6]Strictly speaking, ConQueR can retain in Q' other relations that are not in $rel(Q)$ and contain a "matching attribute" A'_i of some column C_i in $Q(D)$ such that the set of constant values in C_i is contained by the values in A'_i. However, this strategy is likely to produce refined queries with higher dissimilarity values.

records G as a candidate schema subgraph if G contains vertices corresponding to all relations in S_R. ConQueR also expands G into larger subgraphs G' by adding one edge that connects a vertex V in G with a neighbor of V that is currently not in G, and places the resultant graph G' into $\mathcal{QG}$. ConQueR constraints the maximum number of vertices in a candidate subgraph not to exceed some threshold value for the refined queries to be meaningful[7]. Finally, ConQueR ranks the candidate schema subgraphs in the increasing order of their dissimilarities.

Generating refined queries. Consider the general case where refined queries Q' are to be generated for a schema that is different from that of Q and involves a set of relations R and a set of join predicates $\mathcal{J}$. ConQueR first rewrites the input why-not question (S, C) into an equivalent question as follows. ConQueR transforms the why-not question into (S', C), where $S' = Q(D) \cup S$ (i.e., tuples in $Q(D)$ are also considered as why-not tuples), and assume that the input query returns empty result. The transformed why-not question can be processed using the previously discussed techniques as follows. First, ConQueR^s generates a refined query Q' with low dissimilarity value such that $rel(Q') = R$, $join(Q') = \mathcal{J}$, $sel(Q') = \emptyset$, and $proj(Q')$ contains the corresponding attributes in $proj(Q)$. Note that if $Q'(D)$ cannot account for all the why-not tuples in S', then there are no refined queries for this schema and ConQueR will consider another query schema for possible refined queries.

If Q' is a refined query, ConQueR^p will try to enhance the precision of Q' by adding additional selection predicates. Assume there are n why-not tuples in S'. Similar to the discussion in Section 5.2.2, ConQueR^p derives the set of skylines tuples SL_i of each M_i w.r.t. all attributes in $rel(Q')$ [8]. ConQueR^p then enumerates different refined queries Q' that correspond to different subsets $M' \subseteq \cup_{i=1}^{n} SL_i$ of matching tuples, where each M' consists of one tuple from each of M_i. When the number of attributes in $rel(Q')$ is high,

[7]In our implementation, the threshold value is set to be 5 by default.

[8]ConQueR^p considers to add the selection predicates in the form "$A \leq v$" to reduce the search space in this case. Thus, a tuple t_x dominates another tuple t_y if (1) $t_x.A \leq t_y.A$ for every attribute A in $rel(Q')$, and (2) at least one of the inequalities in (1) is strict

the number of skyline tuples in each M_i becomes large. This event leads to high computation for ConQueR^p to enhance $sel(Q')$. To avoid this computational issue, ConQueR^p finds k-dominant skyline tuples in each M_i instead of finding all skylines in M_i. A tuple t_x is said to k-dominate another tuple t_y if there are k dimensions in which t_x is better than or equal to t_y and is better in at least one of these k dimensions. A tuple that is not k-dominated by any other tuples is in the k-dominant skyline [8]. In our experiment, we set k to be $2/3$ times the number of attributes in $rel(Q')$.

Example 5.4 *Consider again query Q_1 in Example 5.1 and another why-not question $S = \{t_w\}$ where $t_w = $ ("E"). Here,* ConQueR *is unable to derive any refined query with the same schema as Q_1, since t_w does not have any matching tuples in $Q^*_{\emptyset}(D)$. To generate refined queries with a different schema from Q_1,* ConQueR *transforms the why-not question to become $S' = \{$("A"), ("E")$\}$, and is now able to derive a refined query Q'_3 that involves the join between Player and Playoff: SELECT name FROM Player, Playoff WHERE Player.pID = Playoff.pID AND pts $\leq$ 2029.* □

In the event that ConQueR cannot find any SPJ refined queries, ConQueR will resort to derive SPJU refined queries Q' of the form: $Q' = Q$ *union* Q_s, such that Q_s accounts for the why-not tuples in S. To derive Q_s, ConQueR first needs to determine $rel(Q_s)$. Since the why-not tuples in S are essentially contained in a $|proj(Q)|$-column table T, $rel(Q_s)$ must be selected such that for each column C_i in T, there must be a "matching attribute" A'_i in some relation in $rel(Q_s)$ such that the set of constant values in C_i are contained by the values in A'_i. The domain indices technique described in Section 4.3 can be applied here to derive the matching attributes for each column of T. For each potential candidate for $rel(Q_s)$, Q_s is constructed by ConQueR^s as follows: $sel(Q_s)$ is defined to be an empty set, $join(Q_s)$ is defined to be the set of foreign-key join predicates among the relations in $rel(Q_s)$, and $proj(Q_s)$ is defined to be set of matching attributes. If $Q_s(D)$ derived from the resultant query schema (defined by $rel(Q_s)$ and $join(Q_s)$) can account for all the why-not tuples, then the query Q_s produced by ConQueR^s can be further enhanced by ConQueR^p to improve its precision.

5.3 Explaining SPJA Queries

In this section, we explain how ConQueR generates refined queries for SPJA queries. For simplicity and without loss of generality, we assume there is only a single aggregated attribute in $proj(Q)$ based on SUM operator, and we use A_a to denote the attribute in $proj(Q)$ being aggregated and A_{agg} to denote $SUM(A_a)$. We also assume that the domain of A_a contains positive values. Details on how the techniques can be generalized for other cases (e.g., the domain of A_a contains negative values) are given in Section 5.4.

As the examples in Section 1.3 illustrated, ConQueR can handle two types of why-not questions on SPJA queries. In the first basic type of why-not questions, each why-not tuple corresponds to either some existing tuple $t_i \in Q(D)$ or some missing tuple t_i, and the question asks why $t_i.A_{agg}$ is not greater than some value K_i. In the second more complex type of why-not questions, it involves at least two why-not tuples, t_1 and t_2 (which may be existing or missing tuples), and the explanation sought is to clarify on the relationship between their A_{agg} attribute values. For example, if t_1 and t_2 are two existing tuples in $Q(D)$ with $t_1.A_{agg} \leq t_2.A_{agg}$, then the why-not question asks why $t_1.A_{agg}$ is not greater than $t_2.A_{agg}$.

To simplify the presentation, we shall assume that for each why-not tuple t in S, the components corresponding to the non-aggregated values (i.e., group-by attributes) in t all have constant values.

While the processing of why-not questions on SPJ queries requires $Q'(D)$ to contain a single matching tuple from M_i for each why-not tuple $t_i \in S$, the processing for SPJA queries is more complex, as $Q'(D)$ needs to contain a subset of matching tuples from M_i to satisfy the aggregation constraint of each why-not tuple $t_i \in S$.

5.3.1 Basic Why-not Questions

Let us consider the case where Q and Q' have the same query schema, and there is exactly one why-not tuple $S = \{t_1\}$ that is a missing tuple (i.e., $t_1 \notin Q(D)$) and the constraint in C requires $t_1.A_{agg} > K$.

As in Section 5.2.1, we assume that $sel(Q) = \{A_1 \leq v_1, \cdots, A_\ell \leq v_\ell\}$, $\ell \geq 1$. Let v'_i denote the modified value of v_i in Q', for $i \in [1, \ell]$. The definition of $Q^*(D)$ and v_i^{max} in Section 5.2.1 is used here as well. Let J_q denote the subset of tuples in $Q^*_\emptyset(D)$ that are matching the tuples of $Q(D)$; i.e., for every tuple $t_q \in J_q$, there exists one tuple $t \in Q(D)$ such that for every non-aggregated attribute component of t_q, the corresponding component of t has the same value.

Naive ConQueR (ConQuerR$^-$). To motivate the optimizations adopted by ConQueR to process why-not questions on SPJA queries, we first present a simpler variant of ConQueR, denoted by ConQueR$^-$.

For each selection predicate attribute A_i, $i \in [1, \ell]$, let lb_i denote the smallest A_i value among $\{t.A_i | t \in M_1\}$ that satisfies the constraint $\sum_{t \in M_1, t.A_i < lb_i}(t.A_a) \leq K < \sum_{t \in M_1, t.A_i \leq lb_i}(t.A_a)$. It follows that for Q' to be a refined query for the why-not question, we must have $v'_i \geq lb_i$, for $i \in [1, \ell]$. Moreover, for $Q'(D) \supseteq Q(D)$, we must have $v'_i \geq v_i^{max}$, for $i \in [1, \ell]$ as explained in Section 5.2.1.

Thus, based on the above two constraints, ConQueR$^-$ enumerates all potential values for each $v'_i \in V_i$, where $V_i = \{t.A_i \mid t \in M_1 \ \wedge \ t.A_i \geq \max\{lb_i, v_i^{max}\}\}$. Each combination $(v'_1, \cdots, v'_\ell)$ considered corresponds to a potential refined query Q'. Therefore, if (1) $Q'(D)$ can account for all the why-not tuples in S, and (2) $Q'(D) \supseteq Q(D)$; then Q' is a refined query for the why-not question. Note that for $Q'(D) \supseteq Q(D)$, it is necessary that $Q'(D)$ does not contain any tuples in $J_q - Q^*(D)$[9].

Even with the use of constraints, the total number of potential refined queries to be considered, given by $\prod_{i=1}^{\ell} |V_i|$, is rather large. For efficiency reason, ConQueR$^-$ adopts a two-step approach to generate refined queries. In the first step, a heuristic is used to choose a subset A' of selection attributes in $sel(Q)$. In the second step, A' is used to generate the potential refined queries. Thus, the number of refined queries considered

[9]Suppose Q' selected a tuple $t_q \in J_q - Q^*(D)$, and let $t_d \in Q(D)$ be the tuple in $Q(D)$ that corresponds to t_q. Then $t_d.A_{agg}$ in $Q'(D)$ will be greater than $t_d.A_{agg}$ in $Q(D)$; i.e., $Q'(D) \not\supseteq Q(D)$.

	pID	name	team	year	*pts*	*blk*	*stl*	*reb*
t_1	P1	A	GSW	1973	2029	40	100	30
t_2	P2	B	SEA	1994	3000	65	150	181
t_3	P2	B	CHI	1995	2200	50	120	161
t_4	P2	B	LAL	1996	2500	70	110	200
t_5	P4	D	LAL	1995	2300	70	150	150
t_6	P5	E	DEN	2000	1689	35	200	381

$Q^*_\emptyset(D) = Player \bowtie_{pID} Playoff$

Figure 5-3: Example 5.5

is reduced to $\prod_{A_j \in A'} |V_j|$. While this approach improves efficiency, the tradeoff is that the refined queries generated have higher dissimilarity values, since not all the selection attributes in $sel(Q)$ appear in Q'. In ConQueR$^-$, the heuristic for selecting A' uses an input control parameter θ[10] so that $\prod_{A_j \in A'} |V_j|$ is no larger than θ. To minimize the dissimilarity values of the refined queries, ConQueR$^-$ uses a simple greedy heuristic to maximize the number of selected attributes in A' by selecting the attributes A_j in non-descending order of $|V_j|$.

Example 5.5 *Consider a query Q_2 on the Basketball data set that finds players and their total points scored in play-off games that satisfy some conditions on their block and steal statistics: SELECT name, SUM(pts) FROM Player, Playoff WHERE Player.pID = Playoff.pID AND blk ≤ 40 AND stl ≤ 100 GROUP BY name. The output contains only one tuple (“A”, 2029). Consider the why-not question $S = \{t_w\}$ with $t_w = (“B”, \$x)$ and $C = \{\$x > 3500\}$, which asks why “B”, with a total score of greater than 3500, is missing from the output.*

ConQueR$^-$ *is able to derive refined queries Q' that have the same schema as Q_2 for this why-not question. The output of the intermediate query $Q^*_\emptyset$ to derive Q' is shown in Figure 5-3. Let A_1 and A_2 denote the two selection predicates blk and stl, respectively. We have $Q^* = \sigma_{blk \leq 40 \wedge stl \leq 100}(Q^*_\emptyset)$. Thus, $Q^*(D) = \{t_1\}$, $v_1^{max} = 40$, and $v_2^{max} = 100$. The set of matching tuples in $Q^*_\emptyset(D)$ for t_w is given by $M_w = \{t_2, t_3, t_4\}$.* ConQueR$^-$ *derives $lb_1 = 65$ and $lb_2 = 120$; therefore, $V_1 = \{65, 70\}$ and $V_2 = \{120, 150\}$.*

[10]In our experiments, we set $\theta = 100000$.

ConQueR$^-$ *generates four candidate refined queries as follows. First,* ConQueR$^-$ *selects the set of attributes $A' = \{A_1, A_2\}$ to be used for the refined queries. Next, based on V_1 and V_2, a candidate refined query is generated corresponding to each of the four combinations of (v'_1, v'_2), where $v'_1 \in V_1$ and $v'_2 \in V_2$. Among these four candidates, the query Q'_2 corresponding to the combination $(65, 120)$, given by: SELECT name, SUM(pts) FROM Player, Playoff WHERE Player.pID $=$ $Playoff.pID$ AND blk ≤ 65 AND stl ≤ 120 GROUP BY name, is not a valid refined query. The reason is that the output of Q'_2, which contains the tuples ("A", 2029) and ("B", 2200), does not account for the why-not tuple t_w. The candidates corresponding to the remaining three combinations are valid refined queries.* □

Optimizations. In this section, we present the optimizations adopted by ConQueR to optimize the generation of refined queries. ConQueR is also based on the two-step approach as ConQueR$^-$, where it first selects a subset of attributes A' followed by using A' to generate potential refined queries. However, ConQueR exploits additional properties to prune away the useless candidate refined queries. Thus, ConQueR is able to generate the same set of refined queries as ConQueR$^-$ more efficiently.

Let $A' = \{A_1, \cdots, A_m\}$ denote the set of attributes selected by the greedy heuristic in the first step, where $|V_1| \leq \cdots \leq |V_m|$. Let $M_1 = \{x_1, x_2, \cdots, x_n\}$, where $x_1.A_1 \leq x_2.A_1 \leq \cdots \leq x_n.A_1$. Let x_s be the "first" tuple in M_1 such that $\sum_{t \in M_1, t.A_1 \leq x_s.A_1} t.A_a > K$ and $\sum_{t \in M_1, t.A_1 \leq x_{s-1}.A_1} t.A_a \leq K$. Observe that for Q' to be a refined query, $Q'(D)$ must contain at least one matching tuple from $\{x_s, \cdots, x_n\}$. Otherwise, the selected matching tuples will not be able to account for the missing why-not tuple t_1. Based on this observation, we can view the collection of candidate refined queries as being partitioned into $(n - s + 1)$ groups $G_s, G_{s+1}, \cdots, G_n$ such that for each refined query Q' in group G_i, the matching tuples in M_1 that are selected by Q' include x_i and a (possibly empty) subset of $\{x_1, \cdots, x_{i-1}\}$.

Thus, ConQueR enumerates the candidate refined queries in $(n - s + 1)$ iterations, where at the j^{th} iteration for $j \in [1, n - s + 1]$, Q' selects the matching tuples from M_1

that contains x_{s+j-1} and a subset of $\{x_1, \cdots, x_{s+j-2}\}$. More specifically, in the j^{th} iteration, $j \in [1, n-s+1]$, the following values of $v'_i, i \in [1, m]$ are being considered:

1. v'_1 is set to $\max\{v_1^{max}, x_{s+j-1}.A_1\}$ to ensure that x_{s+j-1} is selected from M_1 and that $Q'(D) \supseteq Q(D)$.

2. For each $v'_i, i \in [2, m]$, the values considered for v'_i are selected from the set $S_i = \{x_1.A_i, \cdots, x_{s+j-1}.A_i\}$ that must satisfy the following constraints:

   (a) $v'_i \geq lb_i$ to ensure that Q' is a refined query;

   (b) $v'_i \geq v_i^{max}$ to ensure that $Q'(D) \supseteq Q(D)$; and

   (c) $v'_i \geq x_{s+j-1}.A_i$ to ensure that x_{s+j-1} is selected by Q'.

Thus, each combination $(v'_1, \cdots, v'_m)$ considered corresponds to a candidate refined query Q'. The total number of combinations considered by ConQueR is $\sum_{i=1}^{n}(i^{m-1})$ in the worst case. Our experimental results in Section 5.7 showed that the pruning optimization enables ConQueR to be 2 to 10 times faster than ConQueR$^-$.

Example 5.6 *This example reconsiders query Q_2 in Example 5.5 to illustrate how the above optimizations enable* ConQueR *to prune away the invalid candidate refined query generated by* ConQueR$^-$. ConQueR *first derives $M_1 = \{x_1, x_2, x_3\}$, where $x_1 = t_3$, $x_2 = t_2$ and $x_3 = t_4$ such that $x_1.A_1 \leq x_2.A_1 \leq x_3.A_1$. As before, we also have $lb_1 = 65$ and $lb_2 = 120$. The "smallest" tuple x_s that satisfies the aggregation constraint is x_2.* ConQueR *enumerates the candidate refined queries in two iterations as follows.*

In the first iteration, v'_1 is set to 65 and v'_2 is selected from the set $S_2 = \{120, 150\}$. Since $v'_2 \geq \max\{lb_2, x_2.A_2\}$, it results in $v'_2 = 150$.

In the second iteration, v'_1 is set to 70 and v'_2 is selected from the set $S_2 = \{110, 120, 150\}$. We have $v'_2 \geq \max\{x_3.A_2, lb_2\}$; or, $v'_2 \in \{120, 150\}$. Thus, ConQueR *generates only the candidate queries corresponding to the combinations (65, 150), (70, 120), and (70, 150), which is a proper subset of those generated by* ConQueR$^-$. □

5.3.2 Complex Why-not Questions

The techniques presented in the previous section to process basic why-not questions on SPJA queries can be extended to handle the more complex why-not questions as well. Consider a complex why-not question on SPJA queries with $S = \{t_1, \cdots, t_k\}$ and the constraint in C requires that $t_1.A_{agg} < \cdots < t_k.A_{agg}$.

The approach for enumerating candidate refined queries in this case follows the same approach discussed in the previous section except that each V_i is now defined as $V_i = \{t.A_i \mid t \in \mathcal{P} \ \wedge \ t.A_i \geq v_i^{max}\}$, where $\mathcal{P} = M_1 \cup \ \cdots \ \cup \ M_k$.

5.4 ConQueR: Further Extensions

In this section, we present the techniques of `ConQueR` to handle categorical attributes appeared in the selection conditions of the input query, and the extensions of `ConQueR` to derive refined queries in the general cases that are not considered in Sections 5.2 and 5.3.

5.4.1 Handling Categorical Attributes

Consider the scenarios when $sel(Q)$ consists of predicates of both numeric and categorical attributes, including (1) n selection conditions on numeric attributes in the form "$A_i \leq v_i$", and (2) m selection conditions on categorical attributes in the form "$A_j \in S_j$", $i \in [1, n]$ and $j \in [n+1, n+m]$.

SPJ queries. To simplify the presentation, consider an SPJ input query Q and a why-not question (S, C) with $S = \{t_1\}$ and $C = \emptyset$. `ConQueR` will modify the selection conditions of Q into "$A_i \leq v_i'$" and "$A_j \in S_j \bigcup S_j'$", for $i \in [1, n]$ and $j \in [n+1, n+m]$, to account for the why-not question.

Similar to the basic framework, `ConQueR` requires $Q'(D)$ to contain only one tuple from M_1 to account for t_1. For $Q'(D)$ to contain a tuple $\tau \in M_1$, we must have $v_i' =$

$\max\{v_i^{max}, \tau.A_i\}$ and $S'_j = S_j \cup \{\tau.A_j\}$, for $i \in [1, n]$ and $j \in [n+1, n+m]$.

With the presence of categorical attributes, the definition of skyline tuples in M_1 changes slightly as follows. Consider two tuples $t_x, t_y \in M_1$, we say that t_x dominates t_y if (1) $t_x.A_i \leq t_y.A_i$ and $t_x.S_j \subseteq t_y.S_j$, for all $i \in [1, n]$ and $j \in [n+1, n+m]$; and (2) at least one of the inequalities of numeric attributes in (1) is strict or one subset condition on categorical attributes in (2) is a strict subset condition.

SPJA queries. With the presence of categorical attributes, ConQueR performs a two-step heuristic approach by first deriving the candidate refined queries using only numeric attributes based on the techniques in Section 5.3 to derive $sel(Q') = \{A_1 \leq v'_1, \cdots, A_n \leq v'_n\}$. In the second phase, for each derived refined query Q', ConQueR further reduces the imprecision of Q' by inserting the selection conditions of categorical attributes into $sel(Q')$. In particular, let $M_s \subseteq M_1$ be the set of tuples in M_1 that is selected by Q', and S'_j be the set of distinct A_j's values of tuples in M_s, $j \in [n+1, n+m]$. ConQueR then modifies the selection condition on A_j in $sel(Q')$ into "$A_j \in S'_j$".

In a special case when there does not exist any selection predicate of numeric attributes in $sel(Q)$ (i.e., $n = 0$), the heuristic of ConQueR is to modify the selection conditions to select the matching tuples for each why-not tuple one at a time until the aggregation constraints are satisfied. In particular, assume there is exactly one why-not tuple $S = \{t_1\}$ that is a missing tuple (i.e., $t_1 \notin Q(D)$) and the constraint in C requires $t_1.A_{agg} > K$ on the aggregated attribute A_a. Let $M_1 = \{x_1, \cdots, x_n\}$ be the set of matching attributes of t_1. ConQueR selects $x_i \in M_1$ in the *non-increasing order* of their A_a values until the summation of the A_a values of the selected x_i's satisfies the constraint (i.e.,greater than K). Finally, ConQueR modifies the selection conditions to account for the selected x_i's.

5.4.2 Extensions of Explaining SPJ Queries

This section introduces the techniques of ConQueR to solve the general constraints on variables of why-not tuples for SPJ queries. Consider the situation with an input SPJ query Q and the why-not question is (S, C), where $S = \{t_1, \cdots, t_n\}$ and the constraint C requires the comparisons among the variables of some why-not tuples in S.

Example 5.7 *Consider the following SQL query to find the recent high-scoring NBA players and the teams they played for: SELECT P.name, R.team FROM Player P, Regular R WHERE P.pID = R.pID AND R.year >* 2000 *AND R.pts >* 2400*. A user asks why two superstars "Magic Jackson" and "Kareem Abdul-Jabbar" do not appear in the output; furthermore, Jackson and Jabbar played in the same team. The why-not question in this case is* (S, C) *with* $S = \{$*(Jackson,*$\$x$*), (Jabbar,*$\$y$*)*$\}$ *and* $C = \{\$x = \$y\}$. □

To handle this general setting, ConQueR enumerates different refined queries Q' corresponding to different subsets $M' \subseteq \cup_{i=1}^{n} M_i$ of matching tuples, where each M' consists of one tuple from each M_i, $i \in [1, n]$. With a candidate refined query Q', ConQueR determines the matching tuples in $Q'(D)$ for each why-not tuple, and returns Q' as a refined query if the matching tuples selected by Q' satisfy the imposed constraints in C.

Example 5.8 *Continuing with Example 5.7, assume that the sets of matching tuples for "Jackson" and "Jabbar" are* $M_1 = \{t_1, t_2\}$ *and* $M_2 = \{t_3\}$*; where the (year, pts, team) values of* t_1*,* t_2 *and* t_3 *are* (2000, 2500,*"DEN"*), (2002, 2000,*"LAL"*) and* (1999, 3000,*"DEN"*), correspondingly. By selecting* $t_1 \in M_1$ *and* $t_3 \in M_2$ *to account for Jackson and Jabbar,* ConQueR *derives a refined query* Q'*: SELECT P.name, R.team FROM Player P, Regular R WHERE P.pID = R.pID AND R.year ≥* 1999 *AND R.pts >* 2400*. The refined query* Q' *is a valid one, since the matching tuples of Jackson and Jabbar have the same "team" attribute values.* ConQueR *also selects* $t_2 \in M_1$ *and* $t_3 \in M_2$ *to cover the why-not tuples; however, the refined query* Q'' *corresponding to this pair of selected tuples is not appropriate, since the matching tuples selected by* Q'' *have different "team" attribute values.*

□

5.4.3 Explaining SPJU Queries

In this section, we explain how ConQueR explains why-not questions for an SPJU input query Q in the form Q_1 *union* $\cdots$ *union* Q_k, where each Q_i is an SPJ query and the select-clauses of Q_i are union-compatible. To simplify the presentation, we first consider the why-not question with one why-not tuple: $S = \{t_1\}$ and $C = \emptyset$, and generalize ConQueR for multiple why-not tuples at the end of this section.

ConQueR first derives an intermediate query $Q^{'}_{int,i}$ corresponding to the sub-query Q_i, for $i \in [1, k]$. Let M_i denote the set of matching tuples of t_1 in each $Q^{'}_{int,i}$. It is possible that $M_i = \emptyset$ for some $i \in [1, k]$. ConQueR then derives the set of "skyline tuples" SL_i from each M_i using the procedure in Section 5.2. To account for the why-not tuple t_1, ConQueR needs to select only one tuple from $\cup_{i=1}^{k}(SL_i)$. Therefore, each tuple in each SL_i corresponds to one candidate refined query, which ConQueR computes their dissimilarity and imprecision to derive the skyline refined queries.

Handling multiple why-not tuples. Assume there are n why-not tuples $S = \{t_1, \cdots, t_n\}$. Let $M_{i,j}$ denote the set of matching tuples of t_i in the intermediate query $Q^{'}_{int,j}$, for $i \in [1, n]$ and $j \in [1, k]$. ConQueR also computes the set of skyline tuples $SL_{i,j}$ from each $M_{i,j}$. Next, ConQueR^s enumerates different refined queries Q' corresponding to different subsets $M' \subseteq \cup_{i=1}^{n} \cup_{j=1}^{k} (SL_{i,j})$ of matching tuples, where each M' consists of one tuple from each of $\cup_{j=1}^{k}(SL_{i,j})$, $i \in [1, n]$, to account for each why-not tuple t_i.

5.4.4 Extensions of Explaining SPJA Queries

This section presents the extensions of ConQueR to find refined queries for an SPJA input query Q.

Multiple aggregated functions. Assume there are m aggregated attributes B_i's, and the why-not question (S, C) consists of one why-not tuple with $S = \{t_1\}$ and $C = \{t_1.B_i > K_i\}$ for some constant K_i, $i \in [1, m]$.

The techniques of ConQueR for the basic case are applied here as well, except that the lower bound for an attribute A in $sel(Q)$ is set to be the maximum value of all the lower bounds lb_j's with $j \in [1, m]$; where lb_j is defined as $\sum_{t \in M_1, t.A < lb_j}(t.B_j) \leq K_j < \sum_{t \in M_1, t.B_j \leq lb_j}(t.B_j)$.

Negative domain. In case the domains of aggregated attributes contain negative values, the basic framework of ConQueR remains the same, except that the lower bound lb_i for each selection predicate attribute A_i is not used any more. Therefore, the derivation of each V_i's set becomes $V_i = \{t.A_i \mid t \in M_1 \ \wedge \ t.A_i \geq v_i^{max}\}$.

COUNT/AVG operator. When the operator on an aggregated attribute A_a is COUNT, ConQueR creates a "virtual" aggregated attribute A_v with a domain value of 1 for all tuples. The constraint on $COUNT(A_a)$ is now converted into the corresponding constraint on $SUM(A_v)$.

Similarly, consider the scenarios when the operator on the aggregated attribute A_a is AVG, ConQueR changes the domain values of A_a and the constraints correspondingly. Specifically, assume the constraint requires $AVG(A_a) > K$. For every tuple $t \in Q^*_{\emptyset}(D)$, ConQueR replaces $t.A_a$ by $t.A_a - K$, and the constraint on the why-not tuple by $SUM(A_a) > 0$.

5.5 Alternative Approach: TALOS$^+$

In this section, we present an alternative approach to generate refined queries for explaining why-not questions that is based on extending TALOS, which is designed for the first variant of QBO to derive instance-equivalent queries.

Recall that given a query Q on a database D, the goal of TALOS is to generate query-based characterizations of the query result $Q(D)$ by deriving instance-equivalent queries (IEQs) Q'. Two queries Q and Q' are defined to be instance-equivalent w.r.t. a database D if their results on D are equal; i.e., $Q(D) = Q'(D)$. TALOS generates instance-equivalent queries Q' for Q on D by considering various query schema for Q' based on the $proj(Q)$ and $join(Q)$. For each candidate schema, TALOS can easily determine

$rel(Q')$, $join(Q')$, and $proj(Q')$. In contrast to ConQueR which uses a constraint-based approach to derive $sel(Q')$, TALOS uses a classification-based approach to determine $sel(Q')$ by constructing decision trees. By enumerating different decision trees to generate different sets of selection predicates for $sel(Q')$, different IEQs Q' are derived for Q. The framework of TALOS is described in details in Chapter 3.

We have extended TALOS to generate refined queries for explaining why-not questions. We refer to this extended approach as TALOS$^+$. The basic idea of TALOS$^+$ is to treat $Q(D)$ together with the why-not tuples as the output result of some query Q', and apply TALOS to derive the IEQs for Q'. A key challenge in extending TALOS, which is a precision-oriented approach, to TALOS$^+$ is the modification of the data classification step to construct "linear" decision trees so that the refined queries generated are more similar to the input queries. In addition, TALOS$^+$ needs to handle the new semantics imposed on SPJA queries.

In the following discussions, we describe how TALOS$^+$ derives the refined queries for the SPJ and SPJA input queries in Sections 5.5.1 to 5.5.3. Finally, we discuss how to use TALOS$^+$ and ConQueR for deriving IEQs and explaining why-not questions in Section 5.5.4.

Notations. The definitions of $Q^*_\emptyset(D)$, $Q^*(D)$, M_i described in Section 5.1.3, and J_q described in Section 5.3.1 are used here as well. At a high level, $Q^*_\emptyset(D)$ is the join result of joining all relations in $rel(Q)$ using the join predicates in $join(Q)$. $Q^*(D)$ is the subset of tuples in $Q^*_\emptyset(D)$ that satisfy $sel(Q)$. M_i is the subset of tuples in $Q^*_\emptyset(D)$ that are the matching tuples of the why-not tuple t_i. J_q is the subset of tuples in $Q^*_\emptyset(D)$ that are matching the tuples of $Q(D)$.

5.5.1 Explaining SPJ Queries

For simplicity and without loss of generality, we discuss the techniques of TALOS$^+$ to derive refined queries that have the same query schema with the given SPJ query Q to explain the why-not question (S, C) with $C = \emptyset$.

Let $Q'(D) = Q(D) \cup S$, and assume that $Q'(D)$ contains k tuples. TALOS$^+$ partitions $Q^*_\emptyset(D)$ into $(k+1)$ disjoint subsets: $Q^*_\emptyset(D) = J_0 \cup J_1 \cup \cdots \cup J_k$; where each subset J_i, $i > 0$, contains the matching tuples of the i^{th} tuple of $Q'(D)$. TALOS$^+$ needs to construct a "linear" decision tree for the selection condition of the derived refined query to be in the conjunctive form; the objective is to make the refined queries more similar to the input query. For this task, TALOS$^+$ builds a decision tree DT with the root node N containing all tuples in $Q^*_\emptyset(D)$, and modifies the process of finding the optimal node splits in the decision tree construction as follows.

Consider the set of tuples $Q^*_\emptyset(D)$ in the root node N to be split into two child nodes N_1 and N_2 based on a value v of a numeric attribute A. Each subset J_i of $Q^*_\emptyset(D)$ is partitioned into two subsets: $J_{i,1}$ (in node N_1) and $J_{i,2}$ (in node N_2); where a tuple $t \in J_i$ is partitioned in $J_{i,1}$ iff $t.A \leq v$. TALOS$^+$ needs to ensure that either N_1 or N_2 will contain all tuples of J_i that will be assigned positive tuples. Otherwise, TALOS$^+$ needs to recursively split both N_1 and N_2; thus, the refined queries will not be in the conjunctive form. To account for this requirement, TALOS$^+$ considers the following two cases, and selects the smaller $Gini(N_1, N_2)$ value in these cases as the optimal Gini index:

(C1) All tuples of $J_{i,3-j}$ are labeled negative, and all tuples of $J_{i,j}$ are labeled positive, for $i \in [1, k]$ and $j \in \{1, 2\}$;

(C2) All tuples of $J_{i,3-j}$ are labeled negative, and exactly-one tuple in each $J_{i,j}$ is labeled positive, for $i \in [1, k]$ and $j \in \{1, 2\}$.

If the optimal value of $Gini(N_1, N_2)$ is due to case (C1) with $j = j_{sat}$ where $j_{sat} = 1$ or $j_{sat} = 2$, then all tuples of $J_{i,j_{sat}}$ are assigned positive labels and all tuples of $J_{i,3-j_{sat}}$ are assigned negative labels, for every $i \in [1, k]$. In contrast, if the optimal value of $Gini(N_1, N_2)$ is due to case (C2), then all tuples of $J_{i,3-j_{sat}}$ are assigned negative labels and the semantic to select *exactly-one* tuple in each $J_{i,j_{sat}}$, $i \in [1, k]$, is propagated to the process of finding the optimal node split at the child node $N_{j_{sat}}$. With the exactly-one semantics, the process of finding the optimal node split can be optimized by the techniques similar to what we have explained so far except that TALOS$^+$ needs to consider only case (C2).

The optimality of the node split computed by TALOS$^+$ is proven in the same way as TALOS, which is presented in Appendix D.

5.5.2 Explaining Basic SPJA Queries

For simplicity and without loss of generality, we assume that there is exactly one why-not tuple $S = \{t_1\}$ which is a missing tuple (i.e., $t_1 \notin Q(D)$), and the constraint in C requires that $t_1.A_{agg} > K$ with A_a denotes the attribute in $proj(Q)$ that is being aggregated; i.e., $A_{agg} = SUM(A_a)$. To simplify the presentation, we discuss the scenarios when TALOS$^+$ derives refined queries that have the same query schema with Q.

While the processing of why-not questions on SPJ queries requires $Q'(D)$ to contain at-least one single matching tuple from M_i for each why-not tuple $t_i \in S$, the processing for SPJA queries is more complex, as $Q'(D)$ needs to contain a subset of matching tuples from M_i to satisfy the aggregation constraint of each why-not tuple $t_i \in S$.

For $Q'(D) \supseteq Q(D)$, similar to ConQueR, TALOS$^+$ labels tuples of $Q^*(D)$ as positive, and tuples in $J_q - Q^*(D)$ as negative. For the why-not tuple t_1, TALOS$^+$ must assign a subset of tuples $M_s \subseteq M_1$ as positive tuples such that the summation of the A_a's values of records in M_s is greater than K. To account for this constraint, TALOS$^+$ also modifies the process to find the optimal splitting conditions in the decision tree construction as follows.

Consider the set of tuples $Q^*_\emptyset(D)$ in the root node N of the decision tree DT to be built. Assume node N is being split into two child nodes N_1 and N_2 based on a value v of a numeric attribute A_{split}. With the splitting on A_{split} and v, M_1 is also partitioned into two subsets: $M_{1,1}$ (in node N_1) and $M_{1,2}$ (in node N_2); where a tuple $t \in M_1$ is partitioned into $M_{1,1}$ iff $t.A_{split} \leq v$. As before, for the derived refined query Q' to be in the conjunctive form, TALOS$^+$ needs to ensure that either N_1 or N_2 will contain all positive tuples. Assume that all bound positive tuples (i.e., the positive tuples belong to $Q^*(D)$) are partitioned into N_1. TALOS$^+$ computes the Gini index of the following two cases, and selects the smaller value among these two cases as the optimal Gini index:

(A1) All tuples in $M_{1,2}$ are labeled negative, and tuples in a subset $M_s \subseteq M_{1,1}$ are labeled positive such that: (1) $\sum_{t \in M_s}(t.A_a) > K$, and (2) the number of tuples in M_s is *maximized*. Let $M_{1,1} = \{x_1, x_2, \cdots, x_m\}$ where $x_1.A_a \geq x_2.A_a \cdots \geq x_m.A_a$. Let x_f be the "last" tuple in $M_{1,1}$ such that: $\sum_{t \in M_{1,1}, t.A_a \geq x_{f-1}.A_a}(t.A_a) > K$, and $\sum_{t \in M_{1,1},(t.A_a) \geq x_f.A_a}(t.A_a) \leq K$. It derives that $M_s = \{x_1, x_2, \cdots, x_{f-1}\}$.

(A2) All tuples in $M_{1,2}$ are labeled negative, and tuples in a subset $M_s \subseteq M_{1,1}$ are labeled positive such that (1) $\sum_{t \in M_s}(t.A_a) > K$, and (2) the number of tuples in M_s is *minimized*. Let $M_{1,1} = \{x_1, x_2, \cdots, x_m\}$ where $x_1.A_a \geq x_2.A_a \cdots \geq x_m.A_a$. Let x_f be the "first" tuple in $M_{1,1}$ such that: $\sum_{t \in M_{1,1}, t.A_a \geq x_{f-1}.A_a}(t.A_a) \leq K$, and $\sum_{t \in M_{1,1},(t.A_a) \geq x_f.A_a}(t.A_a) > K$. It derives that $M_s = \{x_1, x_2, \cdots, x_f\}$.

After the optimal Gini value is computed, all the free tuples will necessarily become bounded; i.e., the tuples in the derived subset M_s are assigned positive labels whereas tuples in $M_{1,1} - M_s$ as well as in $M_{1,2}$ are assigned negative labels.

The optimality of the node split computed by TALOS$^+$ is proven in the same way as TALOS, which is presented in Appendix D.

5.5.3 Explaining Complex SPJA Queries

For simplicity and without loss of generality, consider a complex why-not question on SPJA queries with $S = \{t_1, \cdots, t_k\}$, and the constraint C requires $t_1.A_{agg} < \cdots < t_k.A_{agg}$, with A_a denotes the attribute in $proj(Q)$ that is being aggregated; i.e., $A_{agg} = SUM(A_a)$. To construct a linear decision tree, TALOS$^+$ also modifies the process to find the optimal node split in the decision tree construction process as follows.

Consider the set of tuples $Q_\emptyset^*(D)$ in the root node N of the decision tree DT to be built. Assume node N is being split into two child nodes N_1 and N_2 based on a value v of a numeric attribute A_{split}. With the splitting on A_{split} and v, each M_i is partitioned into two subsets: $M_{i,1}$ (in node N_1) and $M_{i,2}$ (in node N_2). For the derived refined query Q' to be in the conjunctive form, TALOS$^+$ needs to ensure that either N_1 or N_2 must contain all positive tuples. Assume that all bound positive tuples are partitioned into N_1. TALOS$^+$ will label all tuples in $M_{i,2}$ as negative, and choose the smaller $Gini(N_1, N_2)$ value between the following two cases as the optimal Gini value:

(G1) TALOS$^+$ *maximizes* the number of selected tuples in N_1 by first maximizing the number of selected tuples in $M_{n,1}$. After selecting tuples in $M_{n,1}$, TALOS$^+$ finds a maximum number of tuples in $M_{n-1,1}$ so that the summation on A_a values of selected tuples in $M_{n-1,1}$ is less than the summation of selected tuples in $M_{n,1}$. TALOS$^+$ continues to select tuples for other sets $M_{i,1}$, for $i \in [n-2, 1]$, with the objective to maximize the number of selected tuples in these sets. TALOS$^+$ derives $M_{i,1}$ in a similar way to what we have described for case (A2) in Section 5.5.2.

(G2) TALOS$^+$ *minimizes* the selected tuples in N_1 by first finding a minimum number of selected tuples in $M_{1,1}$, and then deriving the minimum number of selected tuples in $M_{2,1}$ so that the summation on A_a values of tuples in $M_{2,1}$ is greater than the summation of selected tuples in $M_{1,1}$. TALOS$^+$ continues this process to derive the selected tuples for other $M_{i,1}$, with $i \in [3, n]$.

The node split procedure of TALOS$^+$ in this section is a heuristic solution.

5.5.4 ConQueR vs. TALOS

In this work, we have introduced two classes of algorithms, named TALOS [11] and ConQueR that solve the two settings of QBO. In particular, TALOS is a precision-oriented approach that derives an IEQ Q' of an input query Q such that $Q'(D)$ is as *precise* as possible (compared with $Q(D)$). ConQueR is a similarity-oriented approach that derives a refined query Q'' for a given input query Q and a why-not question (S, C) such that Q'' is as *similar* as possible to Q. The issue is whether we can apply ConQueR to generate IEQs, as well as applying TALOS to generate refined queries for explaining why-not questions. It turns out that both TALOS and ConQueR can be applied for the two settings of QBO.

Essentially, it is possible to apply ConQueR for the first variant of QBO to derive alter-

[11] We use TALOS to refer to both the basic framework and its derivations including REQUERE and TALOS$^+$.

native characterizations of a given result table $Q(D)$ by formulating a why-not question (S, C) where $S = Q(D)$ (i.e., all tuples in $Q(D)$ are considered as why-not tuples) and $C = \emptyset$, and treating the input query to return empty result. ConQueR will generate refined queries Q' such that $Q'(D) = Q(D)$; the refined queries can be considered as instance-equivalent queries of $Q(D)$ in this case. Correspondingly, it is also possible to apply TALOS for explaining why-not questions by treating $Q(D)$ together with the why-not tuples as the output result of some query Q'. We then apply TALOS to derive the IEQs of Q'; the IEQs in this case can be considered as the refined queries to explain why the set of tuples is missing from $Q(D)$.

Since each technique is tailored for the specific purpose of each variant of QBO, TALOS should be used for the first variant of QBO rather than ConQueR. The reason is that ConQueR is a similarity-driven approach; thus, the instance-equivalent queries Q' derived by ConQueR are quite similar to Q and might contain irrelevant tuples. The queries derived by ConQueR, therefore, might not be too meaningful to give alternative characterizations of tuples in the query results.

At the other extreme, ConQueR should be used for explaining why-not question rather than TALOS, since TALOS is a more precision-oriented approach, the queries generated can be rather different from the input query. In some applications, it may not be too meaningful to explain missing tuples using refined queries that are very different from the input query. Moreover, the performance of TALOS is also slower than ConQueR due to its costly data classification step.

Figure 5-4 visualizes the comparisons of refined queries returned by TALOS and ConQueR in term of the similarity and the precision metrics, which guide the usage of TALOS and ConQueR in each of the two settings of QBO. Our experimental studies in Section 5.7 have validated the trend of ConQueR and TALOS shown in Figure 5-4. It is also interesting to design a hybrid approach that combines the advantages of both ConQueR and TALOS to give users more flexibility to control the precision and the similarity metrics of the returned queries, which future studies can undertake.

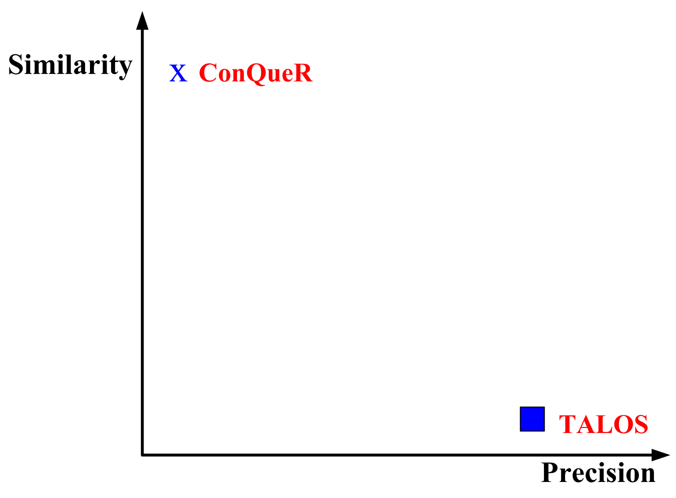


Figure 5-4: Refined queries returned by TALOS and ConQueR

5.6 Implementation of ConQueR

This section presents the implementation details of ConQueR and the comparisons between ConQueR and TALOS in terms of the running time and the space complexity.

5.6.1 Implementation

Similar to TALOS, ConQueR is also implemented at the application level and interacts with the DBMS by issuing relevant SQL queries. For simplicity and without loss of generality, we discuss the implementation details of ConQueR to derive refined queries to explain a why-not question (S, C) for an input SPJ query Q w.r.t. a database D in Algorithm 6. All the steps in Algorithm 6 are self-explained. In the following, we describe how ConQueR utilizes the join indices techniques, which are also used in TALOS, to improve its performance.

Optimization with Join Indices. If the join conditions corresponding to edges in G are foreign-key joins, then the optimization of using the join indices introduced in TALOS (Section 3.3) can be applied here as well. Specifically, ConQueR actually does not compute $Q^*_\emptyset(D)$ but instead computes a hub table J_{hub}, which is the result of the joins among the appropriate join indices corresponding to the edges in G. From J_{hub}, ConQueR also

Algorithm 6: ConQueR(Q, D, S, C)

```
1  Compute sets of core relations of $Q$;
2  foreach set of core relations $\mathcal{R}$ do
3      Enumerate schema subgraphs $G$ containing $\mathcal{R}$ in the (approximately)
       increasing order of their dissimilarity metrics;
4      foreach schema subgraph $G$ do
5          Compute $Q^*_\emptyset(D)$ corresponding to $G$ by joining the relations in $G$ using the
           join conditions corresponding to the edges in $G$;
6          Derive $J_q$ and $M_i$;
7          $exist \leftarrow$ Compute refined queries;
8          if $exist = true$ then
9              return;
10         endif
11     end
12 end
```

derives the mapping tables M_i between each relation $R_i \in G$ and J_{hub}. Each M_i links each record r in R_i to the set of records in J_{hub} that are related to r: for each record r in R_i, there is one record in M_i of the form (j, S), where j is the row identifier of r, and S is a set of row identifiers representing the set of records in J_{hub} that are created from r.

ConQueR also builds a set of attribute lists AL_A for each attribute $A \in R$ with $R \in G$. Each attribute list AL_A, where $A \in R$, is a two-column table (val, row) of the same cardinality as R. Each record $r = (v, i)$ in AL_A corresponds to the i^{th} tuple t in R, and $v = t.A$. Different from TALOS, ConQueR does not necessarily sort order AL_A for computational efficiency.

5.6.2 Complexity Analysis

For simplicity and without loss of generality, we analyze the running time and the space complexity of ConQueR to derive the refined queries for an SPJ input query Q and a set of why-not tuples $S = \{t_1\}$ w.r.t. the schema subgraph G of Q containing n relations $R_1, \cdots,$ R_n. We further assume that the set of projected attributes of Q consists of k attributes A_i, $\cdots, A_k$; where each A_i is an attribute of relation R_i. We also compare the performance of ConQueR with TALOS$^+$. The analysis takes into account the optimizations (i.e., domain indices) in both ConQueR and TALOS$^+$.

Time Complexity. The running time of ConQueR is proportional to the summation of the following four main components, including (T1) to (T4).

(T1) The time $T_{hub}^{conquer}$ to derive $Q_\emptyset^*(D)$, which depends on the join algorithms used inside the DBMS.

(T2) The time $T_{al}^{conquer}$ to derive the set of attribute lists, which is in the order of $O(\sum_{i=1}^{n} |R_i|)$, since ConQueR needs one scan over each relation R_i. Note that ConQueR does not sort order the attribute lists; thus ConQueR saves some computations compared with TALOS in this step.

(T3) The time $T_{cl}^{conquer}$ to derive Q^* and M_i, which is in the order of $O(\sum_{i=1}^{k} |R_i| + |S||Q_\emptyset^*(D)|)$, since ConQueR basically scans the corresponding attribute lists of attributes in $proj(Q)$ and intersects the set of the retrieved tuples.

(T4) The time to derive refined queries, $T_{rq}^{conquer}$, which consists of two components. The first component (ConQueR^s) is to compute skyline tuples in M_1. It is reasonable to assume that M_1 is small enough to be cached in the main memory. ConQueR uses the proposed algorithm in [29] to find a maximal set of vectors for the step of deriving skyline tuples in M_1. The running time of ConQueR^s is, therefore, no greater than $|M_1| \log |M_1|^{d-2}$; where d denotes the number of attributes in $sel(Q)$. The second component (ConQueR^p) basically scans the attribute lists and intersects the sets of row identifiers of tuples satisfying the derived selection conditions. Thus, ConQueR^p runs in $O(\sum_{i=1}^{n} n_i|R_i| + \ell KL)$, where (1) K is the number of skyline tuples derived by ConQueR^s, (2) L is the maximum number of tuples in $Q_\emptyset^*(D)$ that satisfy a modified selection condition derived by ConQueR^p, and (3) ℓ is the maximum number of predicates that are allowed to be inserted into $sel(Q')$. In the worst case, $K = |M_1|$ and $L = |Q_\emptyset^*(D)|$. However, we note that, in practice, K is usually much smaller than $|M_1|$, and L is also smaller than $Q_\emptyset^*(D)$.

Space Complexity. The space complexity of ConQueR is similar to that of TALOS, which is in the order of $((n + 1)|J_{hub}| + \sum_{i=1}^{n} |R_i|)$.

Time	ConQueR	TALOS
Hub Table	$T_{hub}^{conquer}$	T_{hub}^{talos}
Attribute Lists	$O(\sum_{i=1}^{n} \lvert R_i \rvert)$	$O(\sum_{i=1}^{n} n_i \lvert R_i \rvert \log \lvert R_i \rvert)$
Class List	$O(\sum_{i=1}^{k} \lvert R_i \rvert + \lvert S \rvert \lvert Q_{\emptyset}^{*}(D) \rvert)$	$O(\lvert Q(D) \rvert \sum_{i=1}^{k} \log \lvert R_i \rvert + \lvert Q(D) \rvert \lvert J_{hub} \rvert)$
Derive Queries	$O(\sum_{i=1}^{n} n_i \lvert R_i \rvert + \ell K L)$	$O(\ell \sum_{i=1}^{n} n_i \lvert R_i \rvert \lvert Q(D) \rvert)$

Table 5.2: The time complexity comparison of ConQueR and TALOS

Performance Comparison. Intuitively, ConQueR runs more efficiently than TALOS, since while ConQueR mainly manipulates with records of $Q_{\emptyset}^{*}(D)$ corresponding to the why-not tuples, TALOS needs to manipulate the whole set of tuples in $Q_{\emptyset}^{*}(D)$ to trade-off low performance for higher precision refined queries. In the following, we compare the running time of each step of ConQueR with the corresponding one in TALOS; the summaries of these comparisons are provided in Table 5.2.

Both TALOS and ConQueR incur the same computation cost to derive the hub table (in TALOS) or intermediate table (in ConQueR) in the first step. Note that $|Q_{\emptyset}^{*}(D)| = |J_{hub}|$.

In the next step to derive the attribute lists, ConQueR runs faster than TALOS, as TALOS needs to sort the attribute lists whereas ConQueR does not.

In the third step, ConQueR also runs more efficiently than TALOS, since the number of why-not tuples (i.e., $|S|$) is usually smaller than the number of tuples in the query result (i.e., $|Q(D)|$).

Finally, in the last step, ConQueR also runs faster than TALOS, since ConQueR only needs to intersect the sets of row identifiers selected by the derived refined queries. In contrast, TALOS needs to compute the optimal Gini index for each possible splitting attribute by scanning the attribute lists of the involved attributes at each level of the decision tree.

Our experimental results reveal that the performance of TALOS is slower than ConQueR by up to factor of 6 times due to its costly data classification step.

Table	# Tuples
Player	3863
Regular	21376
Playoff	8347
Team	100

(a) Basket ball

Table	# Tuples
order	1500000
partsupp	800000
part	200000
customer	150000
supplier	10000
nation	25

(b) TPCH

Table 5.3: Table sizes (number of tuples)

	Query	Size
Q_1	$\pi_{name}\sigma_{year\geq 2000\wedge pts>2300}\ (Player \bowtie Regular)$	7
Q_2	$\pi_{name,team}\sigma_{year>2000\wedge stl>50\wedge o\_pts\geq 5000}\ (Player \bowtie Playoff \bowtie TeamSeason)$	1
Q_3	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq 1970\wedge pts>2600}\ (Player \bowtie Regular)$	3
Q_4	$\pi_{name,SUM(pts)}\mathcal{G}_{name}\sigma_{year>2000\wedge pts>2300\wedge blk>70}\ (Player \bowtie Regular)$	2
Q_5	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{lost<30\wedge dpts>8000\wedge year\geq 2008}\ (Team \bowtie TeamSeason)$	2
Q_6	$\pi_{part.name}\sigma_{retailprice>2000}\ (part \bowtie partsupp)$	4950
Q_7	$\pi_{supplier.name}\sigma_{acctbal>5000\wedge availqty>3000}\ (supplier \bowtie partsupp)$	4593
Q_8	$\pi_{customer.name}\sigma_{acctbal>9000\wedge totalprice>20000}\ (customer \bowtie order)$	9069

Table 5.4: Test queries for experiments with ConQueR

5.7 Experimental Study

In this section, we evaluate the effectiveness and efficiency of our proposed approach to find explanations for why-not questions. In the first set of experiments (Section 5.7.1), we compare the performance of our constraint-based approach, ConQueR, against the classification-based approach, TALOS$^+$, in terms of the processing efficiency as well as the quality of the derived refined queries. We also validate the efficiency of the pruning optimization in ConQueR for SPJA queries. In the second set of experiments (Section 5.7.2), we compare the effectiveness of our query-refinement based approach of explaining why-not questions against the two existing approaches [9, 22].

We used two data sets for the experiments: the NBA Basketball statistics, and TPC-H data set (with a database size of 1GB). The test data is shown in Table 5.4. The five test queries (Q_1-Q_5) for the Basketball data set and three test queries (Q_6-Q_8) for the TPCH data set are shown in Table 5.4, where the third column indicates the number of tuples

	Why-not questions
W_1	S = {(Rick Barry), (Wilt Chamberlain)}
W_2	S = {(Michael Jordan, WAS)}
W_3	S = {(Kareem Abdul-Jabbar, $\$x$)}, $C = \{\$x > 2000\}$
W_4	S = {(Dwyane Wade,$\$x$), (LeBron James, $\$y$)}, $C = \{\$x < \$y\}$
W_5	S = {(CHI,$\$x$), (DEN, $\$y$), (LAL, $\$z$)}, $C = \{\$x < \$y < \$z\}$
W_6	S = {(coral forest), (chiffon papaya), (lemon dark), (azure beige), (tomato midnight)}
W_7	S = {Supplier4, Supplier50, Supplier60}
W_8	S = {(Customer105155), (Customer90145), (Customer65407), (Customer78322), (Customer82661), (Customer35273), (Customer48008), (Customer101203), (Customer78421), (Customer127777)}

Table 5.5: Why-not questions

	ConQueR^s		ConQueR		TALOS$^+$	
Query	d	i	d	i	d	i
Q_1	2	24	14	6	17	1
Q_2	47	9562	74	696	59	0
Q_3	1	0	1	0	2	0
Q_4	3	0	3	0	12	0
Q_5	3	8	9	0	12	0
Q_6	1	941	1	941	4	941
Q_7	2	428	2	428	5	426
Q_8	2	1849	2	1849	5	1848

Table 5.6: The dissimilarity (d) and the imprecision (i) values of refined queries

in the output of each test query. Table 5.5 shows the why-not questions used for these queries, where the why-not question W_i is asked on query Q_i, $i \in [1, 8]$.

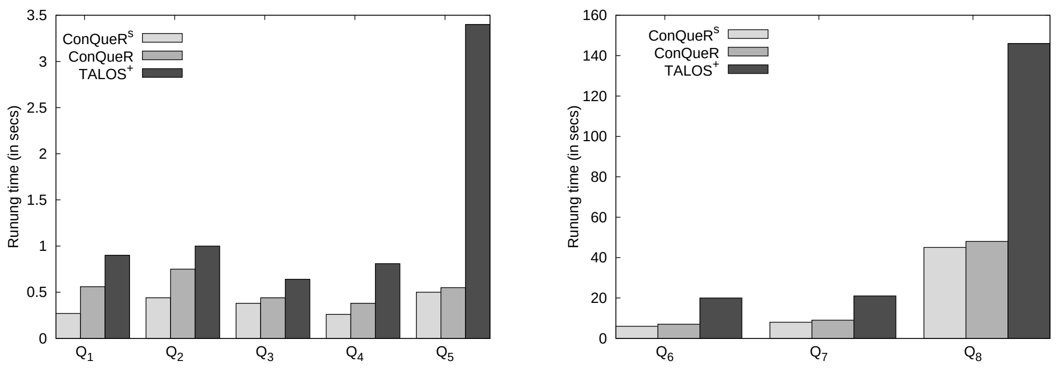
We used MySQL Server 5.0.51 for our database system, and all algorithms were coded using C++ and compiled and optimized with GNU C++ compiler. Our experiments were conducted on a dual-core, 2.33GHz PC running Linux with 3.25GB of RAM and a 250GB hard disk.

5.7.1 Comparing ConQueR & TALOS$^+$

In this section, we compare the performance of ConQueR and TALOS$^+$. We also included the performance of ConQueR^s to understand the tradeoffs between the two key components of ConQueR.

For both ConQueR and TALOS$^+$, we limit the maximum number of selection predicates in refined queries to be 3 times the number of selection predicates in the input query. The time taken to process each why-not question is measured as follows. For ConQueR^s, the time reported refers to the processing time to derive all the refined skyline queries. For ConQueR, the time reported is a sum of two components: (1) the time incurred by ConQueR^s to generate a set of refined skyline queries, and (2) the time taken by ConQueR^p to maximize the precision of each refined query produced by ConQueR^s and output the final skyline refined queries. For TALOS$^+$, the time reported refers to the processing time to generate only the first skyline refined query (i.e., the query corresponding to the first constructed decision tree). The quality of the refined queries are compared in terms of the dissimilarity and the imprecision metrics, where smaller values indicate better quality. Note that if we had measured the total time for TALOS$^+$ to generate all skyline refined queries, the time reported for TALOS$^+$ would have been higher by a factor of 4 to 7 times.

Quality of Refined Queries. Table 5.6 compares the quality of the refined queries. Observe that the refined queries computed by ConQueR^s have the lowest dissimilarity values but the highest imprecision values. At the other extreme, the refined queries generated by TALOS$^+$ have the lowest imprecision values but the highest dissimilarity values. In contrast, the refined queries produced by ConQueR are not only similar to the original queries but also nearly as precise as these generated by TALOS$^+$. For TALOS$^+$, the reason for the high dissimilarity values for its refined queries is that the refined queries can include many selection attributes that are not in the original queries. ConQueR, on the other hand, first uses ConQueR^s to derive refined queries with low dissimilarity values, and then enhances their precision with additional selection predicates. The overall quality of the refined queries generated by ConQueR is, therefore, rather good in terms of both the dissimilarity and imprecision metrics. For some queries (e.g., Q_1, Q_4), although the number of the selection attributes in the refined queries generated by ConQueR and



(a) Basketball (b) TPCH

Figure 5-5: Running time comparisons among ConQueR, ConQueR^s and TALOS$^+$

TALOS$^+$ are nearly the same, the refined queries computed by ConQueR are relatively more similar to the input queries, because ConQueR uses more attributes that appear in the original queries than TALOS$^+$. The imprecision of the refined query for Q_2 by ConQueR^s is much higher than that of ConQueR and TALOS$^+$, since ConQueR^s finds the refined queries on the alternative query schema and does not add any selection predicates into $sel(Q')$ in this case.

Processing Efficiency. The running time performance comparisons are shown in Figures 5-5(a) and (b), respectively, for the Basketball and TPC-H data sets. Since ConQueR^s is only one component of ConQueR, the performance of ConQueR^s is, not surprisingly, better than that of ConQueR. The experimental results show that ConQueR outperforms TALOS$^+$ by a factor of 1.5 to 6 times, indicating the efficiency of the constraint-based approach over the classification-based approach. The classification-based approach incurs a high computation overhead to determine optimal node splits.

Effectiveness of Pruning Optimization. To validate the effectiveness of the pruning optimization in ConQueR for processing SPJA queries, we also compare the performance of ConQueR against ConQueR$^-$.

Table 5.7 compares the number of considered candidate refined queries and the running times of ConQueR and ConQueR$^-$ for the SPJA queries Q_3, Q_4, and Q_5. The results clearly demonstrate the effectiveness of the pruning optimization. For queries Q_3 and Q_4,

Query	# Candidate queries		Running time (s)	
	ConQueR	ConQueR$^-$	ConQueR	ConQueR$^-$
Q_3	18	380	0.44	0.65
Q_4	31	600	0.38	0.70
Q_5	1263	63455	0.55	53.50

Table 5.7: Comparison of ConQueR and ConQueR$^-$

ConQueR is 1.5 to 2 times faster than ConQueR$^-$, while for query Q_5, ConQueR is two orders of magnitude faster than ConQueR$^-$. This huge performance difference is due to the significant pruning of useless candidate refined queries: the number of candidate refined queries considered by ConQueR and ConQueR$^-$ are, 1263 and 63455, respectively.

5.7.2 Comparison of Explanation Models

In this section, we evaluate the usefulness of our proposed query refinement approach to explain why-not questions. We also compare the explanations obtained from the two existing approaches: the approach that is based on identifying the culprit operators that filtered out the missing tuples [9], which we denote by CO, and the approach that is based on database modifications to produce the missing tuples [22], which we denote by DM.

We used the test queries on the Basketball data set (i.e, Q_1 to Q_5 in Table 5.4) and their corresponding why-not questions (i.e., W_1 to W_5 in Table 5.5). Table 5.8 shows the refined queries, denoted by R_i^M, computed for the why-not question W_i on query Q_i using approach M, where $M \in \{conquer^s, conquer, talos^+\}$. The last column in Table 5.8 shows the number of refined queries returned by each method. When ConQueR or ConQueR^s returns more than one refined query, we only report the one that is the most similar to the input query.

Query Q_1 finds the recent high-scoring NBA players. Although some expected well-known superstar players such as "LeBron James" and "Kobe Bryant" are included in the result, other superstar players such as "Rick Barry" and "Wilt Chamberlain" are

	Refined query	Num
$R_1^{conquer^s}$	$\pi_{name}\sigma_{year\geq1965\wedge pts\geq2302}$ $(Player \bowtie Regular)$	1
$R_1^{conquer}$	$\pi_{name}\sigma_{year\geq1965\wedge pts\geq2302\wedge dreb\leq121\wedge asts\geq282\wedge oreb\leq507\wedge weight\geq165}$ $(Player \bowtie Regular)$	1
$R_1^{talos^+}$	$\pi_{name}\sigma_{pts>2345\wedge asts>403\wedge ftm\leq675\wedge gp\leq80\wedge pf\leq286\wedge reb>223}$ $(Player \bowtie Regular)$	1
$R_2^{conquer^s}$	$\pi_{name,team}(Player \bowtie Regular \bowtie TeamSeason)$	1
$R_2^{conquer}$	$\pi_{name,team}\sigma_{ofga\leq6664\wedge hinch\leq6\wedge oto\leq1201\wedge oftm\leq1658}$ $\sigma_{weight\leq210\wedge doreb\leq960\wedge oreb\leq3533\wedge firstseason\leq2003\wedge opf\leq1871}$ $(Player \bowtie Regular \bowtie TeamSeason)$	1
$R_2^{talos^+}$	$\pi_{name,team}\sigma_{fga>1526\wedge o\_reb\leq3312\wedge pf\leq178\wedge weight>165}$ $(Player \bowtie Regular \bowtie TeamSeason)$	1
$R_3^{conquer^s}$	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq1970\wedge pts\geq2596}$ $(Player \bowtie Regular)$	1
$R_3^{conquer}$	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq1970\wedge pts\geq2596}$ $(Player \bowtie Regular)$	1
$R_3^{talos^+}$	$\pi_{name,AVG(pts)}\mathcal{G}_{name}\sigma_{year\leq1971\wedge pts>2637}$ $(Player \bowtie Regular)$	1
$R_4^{conquer^s}$	$\pi_{name,SUM(pts)}\mathcal{G}_{name}\sigma_{year\geq2007\wedge pts\geq2250\wedge blk\geq81}$ $(Player \bowtie Regular)$	2
$R_4^{conquer}$	$\pi_{name,SUM(pts)}\mathcal{G}_{name}\sigma_{year\geq2007\wedge pts\geq2250\wedge blk\geq81}$ $(Player \bowtie Regular)$	2
$R_4^{talos^+}$	$\pi_{Name,SUM(pts)}\mathcal{G}_{name}\sigma_{year\leq2003\wedge first\_season>2002\wedge ftm>202\wedge tpm\leq63}$ $(Player \bowtie Regular)$	1
$R_5^{conquer^s}$	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{lost\leq28\wedge d\_pts\geq8109\wedge year\geq1992}$ $(Player \bowtie Regular)$	1
$R_5^{conquer}$	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{lost\leq28\wedge d\_pts\geq8109\wedge year\geq1992}$ $\sigma_{d\_fgm\leq3139\wedge o\_blk\geq410}$ $(Player \bowtie Regular)$	1
$R_5^{talos^+}$	$\pi_{team,SUM(won)}\mathcal{G}_{team}\sigma_{o\_fga\leq3989\wedge d\_fgm\geq1708\wedge d\_oreb\leq628}$ $(Player \bowtie Regular)$	1

Table 5.8: Refined queries for test queries on Basketball data set

missing from the result. The why-not question W_1 seeks an explanation for these two missing players. The CO approach would have simply identified the selection predicate "*year* $\geq$ 2000" as the reason for Q_1 to have excluded the missing tuples. The DM approach would have suggested several possible ways to modify the data set for the two why-not tuples to be selected by Q_1. For instance, if all the attributes of $sel(Q_1)$ were allowed to be modified, then there will be a total of 224 ways to modify the existing tuples: there are 12 ways to modify the tuples in *Regular* relation to include Barry, and there are 12 ways to modify the tuples in *Regular* relation to include Chamberlain. The refined query computed by ConQueR^s, however, not only implicitly points out that missing tuples are excluded due to the predicate on the *year* attribute, but it also reveals the additional information that the missing players are actually superstars in the 1960's. The refined query computed by ConQueR has higher precision as it further introduces additional selection predicates on attributes such as *weight*, *asts*. The refined query computed by TALOS$^+$ has higher precision but lower similarity compared to the refined query computed by ConQueR. In fact, for the other test queries, although the refined queries computed by TALOS$^+$ have slightly higher precision (relative to those computed by ConQueR), the refined queries are very different from the original queries. For instance, the refined query computed by TALOS$^+$ for Q_4 uses a very different set of attributes from the attributes in $sel(Q_4)$.

Query Q_2 finds the players and the teams that they were playing for when the teams gained a large number of offense points and steal statistics. The why-not question W_2 asks why "Michael Jordan" and his team "WAS" are missing in the result. The CO approach would not be able to generate any explanation for this query, because when Jordan was playing for "WAS", he did not participate in any playoff games; thus, the why-not tuple does not have any matching tuples in the join result of *Player*, *Playoff* and *TeamSeason*. For the DM approach, if the projected attributes were not allowed to be modified, then no explanation can be given for the same reason. Otherwise, there are a total of 13 ways to modify the *team* attribute of the tuples corresponding to Jordan to return the why-not tuple. Our query refinement approach, which can derive refined queries that have different schema from the input query, is able to compute a refined query that involves the relations *Player*, *Regular*, and *TeamSeason*. From this refined query, the user can figure out why ("Jordan", "WAS") was missing from the original query's result: it is due to the fact that Jordan participated in only regular-season games when he was playing for "WAS".

For queries Q_3, Q_4, and Q_5 that are SPJA queries with complex why-not questions, the approach CO is not applicable. The approach DM, which is the most flexible approach, in general has many possible options to modify values in the data set to satisfy the aggregation constraints for these complex why-not questions. In the rest of this section, we will just focus on the explanations computed by `ConQueR`.

Query Q_3 computes the average of the "high points" (defined to be more than 2600 points) scored by players in regular-season games for the period until 1970. The output includes ("Rick Barry", 2775), ("Wilt Chamberlain", 3159) and ("Elgin Baylor", 2719). The why-not question W_3 asks why "Kareem Abdul-Jabbar" with an average high-point score of more than 2000 is missing from the result. The refined query computed by `ConQueR` indicates that the missing tuple will be included if the predicate on pts is modified to become "$pts \geq 2596$". This refined query turns out to be a precise refined query that returns exactly one additional tuple that matches the missing tuple.

Query Q_4 computes the total points scored by players for regular-season games that satisfy the following three conditions: $year > 2000$, $pts > 2300$, and $blk > 70$. The result contains only two tuples: ("Dwyane Wade", 2386) and ("LeBron James", 2304). The why-not question W_4 asks why the total points of James are not higher than that of Wade. The refined query computed by `ConQueR` modifies the three selection predicates as follows: $year \geq 2005$, $pts \geq 2304$, and $blk \geq 66$; and its output now contains (Wade, 2386) and (James, 4782).

Query Q_5 computes the total games won by teams that satisfy the following conditions: $lost < 30$, $dpts > 8000$, and $year \geq 2008$. The result contains two tuples, ("DEN", 108) and ("LAL", 65). The complex why-not question W_5 asks why the team "CHI" is not in the result such that among the three teams, (1) the total games won by "CHI" is the minimum, and (2) the total games won by "LAL" becomes the maximum. The refined query computed by `ConQueR` modifies the predicates as follows: $lost \leq 28$, $dpts \geq 8109$ and $year \geq 1992$; and its output now contains the tuples ("CHI", 57), ("DEN", 108), and ("LAL", 122).

5.8 Summary

In this chapter, we introduced a new paradigm for explaining why-not questions on query results (the second variant of QBO). Our approach, named ConQueR, is based on automatically generating a refined query, whose result includes both the original query's result as well as the user-specified missing tuples. In contrast to the existing explanation models [9, 22], our approach goes beyond merely identifying the "culprit" query operator responsible for the missing tuples, and is useful for applications where it is inappropriate to modify the database to obtain missing tuples. We have proposed novel algorithms to generate good quality refined queries that are not only similar to the original query, but also produce (approximately) precise query results with a small number of irrelevant tuples. Besides the basic SPJ queries, ConQueR can also answer complex why-not questions on SPJ queries with aggregation that involve comparison constraints. Our experimental results demonstrated that ConQueR not only offers a more flexible approach to explain why-not questions, but its constraint-based method of deriving refined queries is also more efficient than the classification-based method of TALOS for the first variant of QBO.

Chapter 6

Instantiation and Evaluation of Partial Queries

It is desirable for database systems to provide novice users with a flexible method to query the systems, while still providing expert users with tools that maximize their productivity. Our third setting of QBO, highlighted in Table 6.1, introduces the concept of partial queries, which works towards addressing this usability issue. We present the definition of partial queries in Section 6.1, and our approaches to solve the two evaluation modes relating to partial queries: (1) evaluation of partial queries (Section 6.2), and (2) instantiation of partial queries (Section 6.3). We show the experimental results of evaluating and instantiating partial queries in Section 6.4. Finally, we summarize our work on partial queries in Section 6.5. Part of the contents and materials in this chapter were previously published in [50].

6.1 Partial Queries

A partial query $Q = (Q_{base}, C_{ans})$ consists of two components: a *base query* Q_{base} and a *set of constraints* C_{ans}. The base query Q_{base} is a conventional relational query that retrieves a set S_{base} of tuples, which serves as the base data for the partial query. Each sub-

QBO Problem	Parameters	
	Input query Q	Given result table T
The first variant	- Q is known - Q is unknown	$T = Q(D)$ T is a set of specific tuples
The second variant	Q is known	$T = Q(D) \bigcup S$ S is a set of tuples that are not present in $Q(D)$
The third variant	**Q is partially specified**	**T is a set of constraints that must be satisfied by the query result of each derived query Q′**

Table 6.1: The Focus of Chapter 6

set $S_{ans} \subseteq S_{base}$ that satisfies all the constraints in C_{ans} is a result of the partial query Q.

The basic aggregation constraint in C_{ans} is a numeric constraint of the form "X *op* c", where X is some expression (to be described), *op* is one of the standard comparison operators ($=, \leq, \geq, <, >$), and c is a non-negative integer constant. The constraints permitted in C_{ans} are of the following six types:

(C1) A **sum constraint** is a constraint on the sum of some attribute A_i in S_{ans}, denoted by $sum(A_i)$ op c.

(C2) A **count constraint** is a constraint on the number of *distinct values* of a subset A' of the attributes in S_{ans}, denoted by $count(A')$ op c.

(C3) A **cardinality constraint** is a constraint on the number of tuples in a query result (or the intermediate result of a subexpression of a query). We denote this constraint by $|Q|$ op c, where Q is either a query or a query's subexpression.

(C4) An **optimization constraint** is specified to maximize/minimize an aggregated value, and there are two forms of optimization depending on whether the aggregated value is bounded. A *bounded optimization constraint* is of the form "opt $(agg(X))$ op c", and an *unbounded optimization constraint* is of the form "opt $(agg(X))$". Here, *opt* is either *minimize* or *maximize*; *agg* is an aggregation operator (sum, count, cardinality); X is either a single attribute (if *agg* is sum or count), a sequence of attributes (if *agg* is count), or a query subexpression (if *agg* is cardinality); and *op* is a non-equality comparison operator. As an example, the constraint $maximize(sum(A_i)) \leq c$ is a bounded optimization constraint on sum, while the constraint $maximize(sum(A_i))$ is an unbounded optimization constraint on sum. Note that C_{ans} can contain at most one optimization constraint.

(C5) A **content constraint** on S_{ans} is of the form $contain(A', V)$, where A' is a subset of attributes in S_{ans}, and V is a set of tuple values (of the same arity as the number of attributes in A'). The constraint requires that the projection of S_{ans} on A' must contain the set of tuples V.

(C6) A **group-by constraint** on S_{ans} is of the form $groupby(A', agg, B')\ op\ c$, where A' and B' are subsets of attributes in S_{ans}, and agg is an aggregation operator (i.e., sum, count, or cardinality). The constraint requires that if S_{ans} is partitioned into groups of tuples having the same values for attribute(s) A', then each group G must satisfy the aggregation constraint $agg(B')\ op\ c$. For the case where agg refers to cardinality aggregation, the B' parameter is unnecessary and is omitted. In addition to this basic form of group-by constraint where the same count/sum constraint (i.e. c) is applied to each group, it is also possible to specify individual count/sum constraint for each group by explicitly listing the desired values of c's using the form: $groupby(A' = v_i, agg, B')\ op\ c_i$. Note that C_{ans} can contain at most one group-by constraint.

We will use PQE to refer to the problem of evaluating partial queries, and PQI to refer to the problem of instantiating partial queries. For simplicity and without loss of generality, in our discussion on PQE, cardinality constraints are treated as a form of sum constraints (i.e., summing on a virtual attribute with a value of 1 for each tuple); therefore, we will not explicitly mention cardinality constraints in Section 6.2. We refer to cardinality constraints in Section 6.3 when we discuss PQI, where the constraints relate to the cardinality of intermediate result sizes.

In this work, we require the domain of the attribute involved in a sum or optimization constraint to be non-negative values. Since both PQE and PQI are generally hard problems, we focus on finding some answer for PQE and PQI and leave the problems of ranking and/or finding top-k answers for partial queries as part of our future work.

Running example. We use a song database for our running example. Part of the schema is illustrated in Figure 6-1, where the key attribute names are shown in bold. The *Song*

title	*genre*	*length*	*filesize*	*artist*	*album*
M1	rock	8	6	A1	AL4
M2	rock	5	3	A1	AL1
M3	rock	4	2	A2	AL1
M4	rock	10	2	A3	AL2
M5	rock	6	5	A4	AL3
M6	rock	5	4	A5	AL3
M7	blues	7	5	A2	AL3

(a) *Song*

album	*sales*	*year*
AL1	100	1980
AL2	75	1980
AL3	70	1975
AL4	120	1985
AL5	100	1990

(b) *Album*

Figure 6-1: Running Example: Song Database D

relation describes information about each song (identified by *title*): the attributes *genre*, *length*, *filesize*, *artist*, and *album* refer to its genre, duration (in minutes), file size (in MB), the artist performing this song, and the album that the song belongs to. The *Album* relation describes information about each album (identified by *album*): the attributes *sales* and *year* refer to the number of albums sold and the year that it was released.

6.1.1 Complexity Results

In this section, we establish the hardness of the problem of evaluation and instantiation of partial queries. The proofs of the following two theorems are given in Appendixes E and F.

Theorem 6.1 *Consider a partial query $Q = (Q_{base}, C_{ans})$ with C_{ans} containing only two count constraints: count$(A_1) = n$ and (2) count$(A_2) = m$, where A_1 and A_2 are attributes of S_{ans}. The problem of evaluating Q is NP-complete.* □

Theorem 6.2 *Consider a partial query $Q = (Q_{base}, C_{ans})$ with C_{ans} contains only one cardinality constraint requiring that $|Q_{base}| = n$. The problem of instantiating Q into a query Q' such that the selection condition of Q' is in the conjunctive form and consists of at most ℓ predicates selected from a set of given predicates is NP-hard.* □

6.2 Evaluating Partial Queries

In this section, we consider the problem of evaluating partial queries. Since some special cases of partial queries have been studied with polynomial [2] or pseudo-polynomial evaluation algorithms [18, 26], partial queries belonging to these specialized classes could be evaluated using these techniques.

However, there remains two open issues. First, there is the question of whether there are other non-trivial special cases of partial queries that are amenable to polynomial/pseudo-polynomial evaluation algorithms. Second, the problem of evaluating general partial queries with arbitrary constraints (sum, count, optimization, group-by, content) has not been addressed to the best of our knowledge.

To address these two questions, we present two evaluation algorithms, `DP` and `Greedy`, respectively. The first algorithm, `DP`, is a pseudo-polynomial algorithm that is designed for evaluating partial queries with any number of sum constraints and at most one of either count, content, or group-by constraint. Our second algorithm, `Greedy`, is a heuristic approach for evaluating general partial queries with any combination of constraints.

6.2.1 Dynamic Programming Approach

We first explain how `DP` evaluates partial queries with multiple sum and a single count constraints. For simplicity and without loss of generality, we explain the evaluation for a partial query Q with the following two constraints on attributes A and B:

- Sum maximization constraint: $maximize(sum(A)) \leq K$, and
- Count constraint: $count(B) = m$

Let the number of distinct B attribute values in S_{base} be ℓ. For simplicity and without loss of generality, let the domain of the B attribute values in S_{base} be $dom(B) = \{1, 2, \cdots, \ell\}$[1].

[1]In general, we can easily map an arbitrary set of ℓ values into the set $\{1, 2, \cdots, \ell\}$.

For each $b \in dom(B)$, let S^b_{base} and $S^{\leq b}_{base}$ defined as in Equations 6.1 and 6.2, respectively. Let $E[1 \cdots \ell, 1 \cdots K]$ be a two-dimensional matrix, where each cell $E[b, V]$ is a boolean value defined in Equation 6.3. Each row $E[b, .]$ is a subset-sum problem that can be solved in $O(K|S^b_{base}|)$. Therefore, the entire matrix E can be constructed in $O(K|S_{base}|)$.

Let $\mathcal{D}[1 \cdots \ell, 1 \cdots m, 1 \cdots K]$ be a three-dimensional matrix, where each cell $\mathcal{D}[b, d, V]$ is a boolean value defined in Equation 6.4.

$$S^b_{base} = \{t \in S_{base} \mid t.B = b\} \tag{6.1}$$

$$S^{\leq b}_{base} = \{t \in S_{base} \mid t.B \leq b\} \tag{6.2}$$

$$E[b, V] = true \text{ iff } \exists\, S \subseteq S^b_{base} \text{ s.t. } \sum_{t \in S}(t.A) = V \tag{6.3}$$

$$\mathcal{D}[b, d, V] = true \text{ iff} \exists\, S \subseteq S^{\leq b}_{base} \text{ s.t. } |\pi_B(S)| = \ d \ \wedge \ \sum_{t \in S}(t.A) = V \tag{6.4}$$

`DP` can find a solution if there exists a maximum value $V_{max} \leq K$ such that (1) $\mathcal{D}[\ell, m, V_{max}] = true$, and (2) for other values $V > V_{max}$, $\mathcal{D}[\ell, m, V] = false$. We have the following recurrence relation:

$$\begin{aligned} &\mathcal{D}[b, d, V] = \mathcal{D}[b-1, d, V] \ \vee \\ \exists\, V' \in [1, V] \ s.t. \ (E[b, V'] = 1 \ \wedge \ &\mathcal{D}[b-1, d-1, V-V'] = 1) \end{aligned} \tag{6.5}$$

The recurrence relation indicates that $\mathcal{D}[b, d, V]$ can be derived from either (1) $\mathcal{D}[b-1, d, V]$ if we do not select any tuples from S^b_{base}, or (2) $\mathcal{D}[b-1, d-1, V-V']$ if we select a subset of tuples S' from S^b_{base} with $\sum_{t \in S'} t.A = V'$.

The computation of each $\mathcal{D}[b, d, V]$ requires at most V look up operations on the corresponding row $E[b, .]$ in the E matrix. Thus, the time to build matrix $\mathcal{D}$ in the worst case is $O(m\ell \sum_{V=1}^{K}(V)) = O(K^2 m\ell)$.

Deriving S_{ans}. In addition to the main matrix $\mathcal{D}$, `DP` uses another matrix $DTrace[\ell, m, K]$ that has the same dimensions with $\mathcal{D}$ to derive S_{ans}. Each cell $DTrace[b, d, V]$ is set to either (1) a value 0 if $\mathcal{D}[b, d, V]$ is derived from $\mathcal{D}[b-1, d, V]$, or (2) a value $V' > 0$ if $\mathcal{D}[b, d, V]$ is derived from $\mathcal{D}[b-1, d-1, V-V']$ and $E[b, V']$.

To derive the set of returned tuples S_{ans}, `DP` first determines the maximum value $V_{max} \leq K$ s.t. $\mathcal{D}[\ell, m, V_{max}] = true$. If $DTrace[\ell, m, V_{max}] = 0$, then S_{ans} is the set of tuples that makes $\mathcal{D}[\ell-1, m, V_{max}] = true$. Otherwise, if $DTrace[\ell, m, V_{max}] = V'$, then S_{ans} is the union of the set of tuples that makes $D[\ell-1, m-1, V_{max}-V'] = true$ and the set of tuples that makes $E[\ell, V'] = true$. The technique to derive a set of tuples that makes $E[\ell, V'] = true$ follows a standard procedure for solving the subset-sum problem. We briefly describe this procedure in the following.

Assume that $S^{\ell}_{base} = \{t_1, \cdots, t_y\}$. To compute $E[\ell, .]$, `DP` builds a two-dimensional matrix $F[1 \cdots y, 1 \cdots K]$ with the following recurrence equation.

$$F[i, V] = F[i-1, V] \vee F[i-1, V - t_i.A] \tag{6.6}$$

`DP` maintains another matrix, denoted as $FTrace[1 \cdots y, 1 \cdots K]$, that has the same dimensionality as F. Each $FTrace[i, V]$ keeps track of how $F[i, V]$ is derived; i.e., $FTrace[i, V]$ is set to either (1) $false$ if $F[i, V]$ is derived from $F[i-1, V]$; or (2) $true$, otherwise.

To find a subset S^{ℓ} of tuples that make $E[\ell, V] = true$, `DP` traces from $FTrace[y, V]$. If $F[y, V] = false$, then S^{ℓ} is the set of tuples that makes $F[y-1, V] = true$. Otherwise, if $F[y, V] = true$, then S^{ℓ} is the union of $\{t_y\}$ and the set of tuples that makes $F[y-1, V - t_y.A] = true$.

The space complexity of `DP` is $O(K|S_{base}| + Km\ell)$ to keep the matrices for the recurrence relations in Equations 6.5- 6.6 in the main memory.

Example 6.1 *Consider a slightly modified Example 1.6 (with smaller constraint values) to retrieve a set of "rock" songs S_{ans} such that (1) $maximize(sum(filesize)) \leq 6$, and (2) $count(artist) = 2$. Figure 6-2 shows a simple encoding of the artist's domain*

	title	*artist*	*filesize*
t_1	M1	A1 (1)	6
t_2	M2	A1 (1)	3
t_3	M3	A2 (2)	2
t_4	M4	A3 (3)	2
t_5	M5	A4 (4)	5
t_6	M6	A5 (5)	4

$S_{base} = \sigma_{genre=\text{“rock”}}(Song)$

Figure 6-2: Example 6.1

values (e.g., “A1” is mapped to 1*) from the Song relation in our running database. To derive* S_{ans}*,* DP *builds a matrix* $\mathcal{D}[5,2,6]$*. According to the recurrence relation in Equation 6.5,* $\mathcal{D}[5,2,6]$ *can be derived from* $\mathcal{D}[4,2,6]$*. However, since* $\mathcal{D}[4,2,6] = 0$*,* $\mathcal{D}[5,2,6]$ *must be derived from the second case. In the second case of Equation 6.5, since* $\mathcal{D}[4,1,6-4] = 1$ *and* $E[5,4] = 1$*, it derives that* $\mathcal{D}[5,2,6] = 1$*.* DP *traces from* $DTrace[5,2,6]$ *to return a set* $S_{ans} = \{t_3, t_6\}$ *as the answer.* □

Approximation version of DP**.** When K and/or ℓ is large, the space required by DP might exceed the available memory. In these cases, DP needs to reduce the space requirement by scaling down the domain values of the attribute used with the sum constraint (i.e., A attribute) by some factor $\mathtt{c_f}$; thus, K will be replaced by $K/\mathtt{c_f}$. The solution of DP is approximate in these cases.

Content constraint. We discuss the adaption of DP to solve PQE when C_{ans} contains any number of sum constraints and a single content constraint next.

For simplicity and without loss of generality, assume that C_{ans} consists of two constraints: (1) $maximize(sum(A)) \leq K$, and (2) $content(B, S_{content})$. We also assume the domain of the B attribute values in S_{base} be $dom(B) = \{1, 2, \cdots, \ell\}$.

The definitions of S^{b}_{base}, $S^{\leq b}_{base}$, and matrix $E[.,.]$ described above are used here as well. DP builds a two-dimensional matrix $D^{content}[1 \cdots \ell, 1 \cdots K]$, where each $D^{content}[b, V]$ is a boolean value to indicate whether there exists a subset $S \subseteq S^{\leq b}_{base}$ such that $\sum_{t \in S}(t.A) = V$.

We have the following recurrence relation.

$$\begin{aligned} D^{content}[b, V] = D^{content}[b-1, V] \ \vee \\ \exists\, V' \in [1, V]\ (E[b, V'] = 1 \text{ and } D^{content}[b-1, V-V'] = 1) \end{aligned} \tag{6.7}$$

To satisfy the content constraint, for each value $b \in S_{content}$, we must select at least one tuple from S^{b}_{base} to insert into S_{ans}. For this constraint, DP requires $D^{content}[b, V]$ to be derived from $D^{content}[b-1, V-V']$ and $E[b, V']$ (the second case of Equation 6.7) and not from the first case, for all $b \in S_{content}$. The reason is that if $D^{content}[b, V]$ can only be derived from $D^{content}[b-1, V]$ (the first case of Equation 6.7), then no tuple in S^{b}_{base} has been selected, which violates the content constraint.

DP can find a solution if there exists a maximum value $V_{max} \leq K$ such that (1) $D^{content}[\ell, V_{max}] = true$, and (2) for other values $V > V_{max}$, $D^{content}[\ell, V] = false$.

Group-by constraint. We discuss the adaption of DP to solve PQE when C_{ans} contains any number of sum constraints and a single group-by constraint. For simplicity and without loss of generality, we assume that C_{ans} consists of the following two constraints: (1) Sum maximization constraint: $maximize(sum(A)) \leq K$, and (2) Group-by sum constraint: $groupby(B, sum, A) \leq K'$.

The definitions of S^{b}_{base} and $S^{\leq b}_{base}$ described above are used here as well. Let $F[1 \cdots \ell, 1 \cdots K']$ be a two-dimensional matrix, where each cell $F[b, V] = true$ if $\exists\, S \subseteq S^{b}_{base}$ such that $\sum_{t \in S}(t.A) = V$. Each row $F[b, .]$ is a subset-sum problem that can be solved in $O(K'|S^{b}_{base}|)$. Therefore, the entire matrix F can be constructed in $O(K'|S_{base}|)$.

DP will build a two-dimensional matrix $D^{gb}[1 \cdots \ell, 1 \cdots K]$, where each $D^{gb}[b, V]$ is a boolean value indicating whether there exists a subset $S \subseteq S^{\leq b}_{base}$ such that: (1) $\sum_{t \in S}(t.A) = V$ and (2) $groupby(B, sum, A) \leq K'$ applied on S is true. We have the following recurrence relation.

$$D^{gb}[b, V] = D^{gb}[b-1, V] \ \vee$$
$$\exists\, V' \in [1, K'] \ (F[b, V'] = 1 \text{ and } D^{gb}[b-1, V-V'] = 1)$$

DP can find a solution if there exists a maximum value $V_{max} \leq K$ such that (1) $D^{gb}[\ell, V_{max}] = true$, and (2) for other values $V > V_{max}$, $D^{gb}[\ell, V] = false$.

6.2.2 Greedy Approach

In this section, we present our second algorithm, denoted by Greedy, which is a heuristic approach for evaluating general partial queries with any combination of constraints. As shown in Theorem 6.1, when there are only two count constraints, the PQE problem is already NP-complete *in the strong sense*. For ease of presentation, our discussion is organized into three cases from the simplest scenario to the most general.

Count Constraints. We first discuss the simplest scenario where all the constraints in C_{ans} are count constraints. For simplicity and without loss of generality, we consider a partial query with two count constraints $count(B_i) = m_i$, $i \in [1, 2]$, where $m_1 \leq m_2$. The heuristics of Greedy bases on the following lemma.

Lemma 6.1 *If there exists a subset $S_{count} \subseteq S_{base}$ that has $count(B_1) = m_1$ and $count(B_2) \geq m_2$, then there exists a subset $S_{ans} \subseteq S_{count}$ that has $count(B_1) = m_1$ and $count(B_2) = m_2$.*

Proof of Lemma 6.1. Given a subset $S_{count} \subseteq S_{base}$ that has $count(B_1) = m_1$ and $count(B_2) \geq m_2$, we first pick m_1 arbitrary tuples in S_{count} that have m_1 distinct B_1 values to put into S_{ans}. The number of distinct B_2's values in S_{ans} is currently not greater than m_1, and therefore is also not greater than m_2, since our assumption is $m_1 \leq m_2$. We then need to insert some tuples from $(S_{count} - S_{ans})$ into S_{ans} to increase the number of distinct B_2's values in S_{ans} into m_2. The task is executed by performing $m_2 - |\pi_{B_2}(S_{ans})|$

steps. In each step, we pick a tuple in $(S_{count} - S_{ans})$ to insert into S_{ans} in such a way that the number of distinct B_2's values in the resultant S_{ans} increases by 1. $\square$

Using Lemma 6.1, `Greedy` derives a set $S_{count} \subseteq S_{ans}$ that has $count(B_1) = m_1$ and $count(B_2)$ is a large as possible. The rationale is that if S_{count} has $count(B_2) \geq m_2$, then we can derive S_{ans} from S_{count} satisfying all the constraints. The details of `Greedy` are as follows.

`Greedy` first partitions S_{base} using the values of B_1 attribute, and performs m_1 iterations to insert m_1 partitions of S_{base} into S_{count}. At each iteration, `Greedy` considers all potential partitions in S_{base}, and chooses the "best" partition to insert into S_{count} such that the resultant S_{count} has the largest number of distinct B_2's values. After m_1 iterations, there are two outcomes. If $|\pi_{B_2}(S_{count})| \geq m_2$, `Greedy` derives S_{ans} that satisfies all the count constraints from S_{count} using Lemma 6.1. Otherwise if $|\pi_{B_2}(S_{count})| < m_2$, `Greedy` returns $S_{ans} = S_{count}$ as an approximation solution that does not satisfy the count constraint on B_2.

The time complexity of `Greedy` is $O(m_1|S_{base}|)$, since `Greedy` uses m_1 iterations and scans all tuples in S_{base} in each iteration.

Count & Sum Constraints. We consider a more complex scenario where there is a combination of count and sum constraints. For simplicity and without loss of generality, we consider a partial query with two constraints: (1) $maximize(sum(A)) \leq K$, and (2) $count(B) = m$.

`Greedy` tries to satisfy the "easier" type of constraints before considering the "harder" constraints. Specifically, `Greedy` considers the constraints in the following order: count constraint, sum constraint, and finally the optimization constraint.

To satisfy the count constraint, `Greedy` can select an arbitrary subset $S_{count} \subseteq S_{base}$ that has $|\pi_B(S_{count})| = m$. The heuristics of `Greedy` is based on the observation that the more tuples that S_{count} has, the more flexibility `Greedy` has to select a subset of tuples from S_{count} to satisfy other constraints. Therefore, `Greedy` will find S_{count} that has the *maximum* cardinalities among all possible S_{count}'s. For this task, `Greedy` partitions tuples in S_{base} based on their B's values, and picks m partitions that have the largest cardinalities to form S_{count}.

To satisfy the sum constraint, `Greedy` partitions tuples in S_{count} based on their B attribute values, and selects the tuple that has the smallest A value in each partition of S_{count} to insert into S_{ans}. If $\sum_{t \in S_{ans}}(t.A) > K$, it implies that any other subsets of S_{count} will not satisfy both count and sum constraint; `Greedy` returns S_{ans} as an approximation result in this case.

Finally, `Greedy` handles the optimization constraint by adding some tuples from $(S_{count} - S_{ans})$ into S_{ans} to increase the summation of the A's value of the selected tuples. This task is a subset-sum problem: select a subset of tuples from $(S_{count} - S_{ans})$ that has $max(sum(A)) \leq K - \sum_{t \in S_{ans}}(t.A)$. It is important to note that we cannot add any tuples from $(S_{base} - S_{count})$ into S_{ans}, since it increases the number of distinct B's values in S_{ans} and thus makes S_{ans} violate the count constraint.

The running time of `Greedy` is $O(|S_{base}| + T_{SSP})$, where T_{SSP} is the running time of the solver for the subset-sum problem in the last step of `Greedy`. In this work, `Greedy` uses the conventional pseudo-polynomial algorithm to solve subset-sum problem; thus, $T_{SSP} = O(K|S_{count}|)$.

General Case. The techniques of `Greedy` described above can be extended to the general case when C_{ans} includes any combination of constraints. Following the "easier-to-harder-constraint" heuristics, `Greedy` will consider the constraints in the following order: (1) content constraints, (2) count constraints, (3) sum constraints, and (4) group by together with optimization constraints. The absence of any constraints (e.g., count) allows `Greedy` to skip the corresponding step(s) (e.g., skip the second step for count constraints).

To simplify the presentation, we assume C_{ans} includes the following four constraints: (1) Content constraint: $contain\ (A', S_{content})$; (2) Count constraint: $count(B) = m$; (3) Sum maximization constraint: $maximize(sum(A)) \leq K$, and (4) Group-by constraint: $groupby(A_{gb}, sum, A) \leq K'$.

First, Greedy satisfies the content constraint in such a way that will ease the constraints considered later. In particular, for each value $t \in S_{content}$, let $P_t \subseteq S_{base}$ be the maximal subset of S_{base} that corresponds to t; i.e., $\pi_{A'}(P_t) = t$. Greedy needs to select at least one tuple in each P_t to put into S_{ans} for t to be present in $\pi_{A'}(S_{ans})$. The heuristic of Greedy is to select the tuple that has the smallest A value in each P_t to put into S_{ans} to ease the sum constraint. If S_{ans} violates some of the remaining constraints, Greedy returns S_{ans} as an approximation solution and terminates at this step.

In the next step, Greedy aims to satisfy the count and then sum constraints in a similar way as described above. More specifically, Greedy partitions S_{base} based on their B's attribute, and inserts $m - |\pi_B(S_{ans})|$ partitions that have the largest cardinality into S_{count}. Greedy then selects the tuple that has the smallest A value in each partition of S_{count} to put into S_{ans}.

Lastly, Greedy considers to insert some tuples from $S_{count} - S_{ans}$ into S_{ans} to satisfy the group-by and optimization constraint at the same time. Since we only have sum constraint(s) and a single group-by constraint in this step, Greedy will apply dynamic programming approach of DP to solve this derived sub-problem.

The time complexity of Greedy is $O(K'|S_{base}| + KK'\ell)$ where ℓ denotes the number of distinct A_{gb} values in S_{base}, since Greedy basically scans S_{base} in the first and second step and uses dynamic-programming approach in the last step.

6.3 Instantiating Partial Queries

In this section, we present our proposed algorithm, named LA for **L**ook **A**head approach, to instantiate partial queries. We focus on the context of generating targeted queries for database testing, where the constraints are all cardinality constraints on the query result or the query's intermediate results (corresponding to subexpressions of the query) [7, 35].

The goal of our approach is to generate concise instantiated queries by modifying the existing predicates' constants or adding a small number of additional predicates.

We first present the techniques for the simple case with a single cardinality constraint in Section 6.3.1, and then generalize the discussion to the general case with multiple cardinality constraints in Section 6.3.2. In Section 6.3.3, we will present an alternative sampling-based approach which is more efficient but may not always find a solution.

For simplicity and without loss of generality, we assume all the attributes in the query schema have numeric domains; our techniques can be applied to categorical attributes as well. We also assume, for simplicity, that the base query Q_{base} is a SPJ query; in practice, our techniques can be applied to any SQL query that has no group-by clause.

6.3.1 Single Constraint

Consider a partial query Q with a single cardinality constraint $|Q| = m$. To simplify the presentation, we assume that the base query Q_{base} does not have any selection predicates. To avoid generating complex instantiated queries with many additional selection predicates, LA uses a threshold parameter, denoted by h_{max}, to control the maximum number of additional selection predicates in an instantiated query Q_{inst}.

The instantiated query Q_{inst} is first initialized to be Q_{base}. LA is based on a greedy heuristics that iteratively adds one selection predicate to Q_{inst} until we have $|Q_{inst}| = m$. At each iteration, LA considers all potential selection predicates "$R_i.A_j\ op\ c$" where R_i is a relation in Q_{inst}, and chooses the "best" (or optimal) selection predicate, denoted by P_{opt}, such that the resultant query's result size is at least m and is minimized; ties are broken arbitrarily. At the end of each iteration, there are three possible outcomes. If $|Q_{inst}| = m$, LA terminates with the required instantiated query. If $|Q_{inst}|$ is reduced (compare with the last iteration) and $|Q_{inst}| > m$, LA continues with the next iteration to pick another selection predicate to be added to Q_{inst}. Otherwise, if $|Q_{inst}|$ remains unchanged, LA either returns Q_{inst} as an approximation solution, or backtracks to the previous iteration to choose the next best selection predicate to replace the last chosen predicate. The rationale behind the greedy of LA is to try to add the least number of predicates to reduce the query result to satisfy the cardinality constraint.

LA introduces another control knob to derive more than one queries when the derived instantiated query does not satisfy users' requirements (e.g., $|Q_{inst}|$ differs largely from m). Basically, LA repeats the same process and restricts the potential selection predicates to involve only the attributes that have not been used in the selection conditions of previous Q_{inst}'s. The rationale is that LA tries to search in a new region to "escape" from the local optimal region that previous instantiated queries are derived from.

Note that although our approach is more general than [35] in that the instantiated queries can have additional selection predicates, our approach can be easily adapted to produce instantiated queries without additional selection predicates by simply restricting the potential selection predicates to involve only the attributes in the base query's selection predicates.

Implementation. The selection of an optimal selection predicate at each iteration can be efficiently implemented using appropriate data structures.

First, for each attribute A in S_{base}, we maintain an attribute list of tuple records of the form (v, i) corresponding to the i^{th} tuple $t \in S_{base}$ where $v = t.A$; the attribute list is sorted in non-descending order of v. With this sorted attribute list for attribute A, the number of tuples in S_{base} selected by a selection predicate can be determined in $O(1)$ using the following observation: for two consecutive tuples (v_1, i) and (v_2, j) in the attribute list of A where $v_1 < v_2$; we have $|\sigma_{A \leq v_2}(S_{base})| = |\sigma_{A \leq v_1}(S_{base})| + 1$. Note that if the size of S_{base} is too large to fit into the main memory, some of the attribute lists have to be kept on the disk.

Second, an array C_R can be maintained in the main memory that has the same size as S_{base}, where each $C_R[i]$ is a boolean value to keep track of whether the i^{th} tuple of S_{base} is still being selected after each iteration. Each $C_R[i]$ is initialized with a true value at the beginning, and is updated at the end of each iteration based on whether the newly selected optimal predicate P_{opt} has pruned away the i^{th} tuple.

Example 6.2 *To give an example of how* LA *derives instantiated queries w.r.t. a single* *cardinality constraint, consider a partial query Q on the Song database with Q_{base} = "SELECT * FROM Album", and $C_{ans} = \{|Q| = 3\}$. Q_{inst} is first initialized to be Q_{base}.* LA *builds attribute lists for year and sales attributes and an array C_R that has the size equal to that of Album relation.*

*In the first iteration, among all selection predicates considered for attributes sales and year, the best predicate is "sales > 75", which reduces the result size to three tuples. Hence, the instantiated query is "SELECT * FROM Album WHERE sales > 75".* □

Complexity. The selection of an optimal selection predicate at each iteration requires at most one scan over each attribute of S_{base}; thus, the time complexity is $O(|S_{base}|n_{attr})$, where n_{attr} denotes the number of attributes in S_{base}. Since the number of iterations is bounded by h_{max}, the time complexity to generate one instantiated query without backtracking is $O(|S_{base}|n_{attr}h_{max})$; and with n backtracks, the time complexity increases to $O(|S_{base}|n_{attr}(h_{max} + n))$. The space complexity is $O(|S_{base}|)$ to keep C_R in the main memory.

6.3.2 Multiple Constraints

In this section, we extend the approach of LA discussed in the previous section to instantiate partial queries having multiple cardinality constraints. We explain LA using a partial query Q with C_{ans} containing k cardinality constraints of the form "$|Q_i| = m_i$", for $i \in [1, k]$, where each Q_i is a subexpression of the base query Q_{base}. In the following, let $S_{base,i}$ be the result of each query subexpression Q_i w.r.t. S_{base}.

The idea of LA is to apply the techniques introduced in Section 6.3.1 to instantiate the queries corresponding to the k subexpressions Q_i of Q (w.r.t. the constraint m_i) sequentially in some sequence.

Consider two query subexpressions Q_i and Q_j where Q_i is a subexpression of Q_j. Since the set of selection predicates in Q_i is a subset of those in Q_j, Q_i is instantiated before Q_j. Otherwise, the instantiation of Q_i after the instantiation of Q_j might add new selection predicates to Q_i's instantiation, and therefore also to Q_j's instantiation, which possibly violates Q_j's constraint. The correct strategy is thus to first instantiate Q_i before Q_j, and to instantiate Q_j such that no selection predicates related to Q_i is introduced.

To determine the order of instantiating subexpressions of Q, LA constructs an *order graph* $\mathcal{G}$, where each subexpression Q_i corresponds to a vertex of $\mathcal{G}$ and there exists a directed edge from Q_i to Q_j if Q_i is a subexpression of Q_j. LA then obtains a *topological ordering* of the vertices in $\mathcal{G}$ corresponding to the order of subexpressions of Q that LA will investigate.

When considering a subexpression Q_j and if $|Q_j| < m_j$, then it is impossible to satisfy the constraint for Q_j, since adding any selection predicates into Q_j will further reduce the size of Q_j. Therefore, in the process of instantiating queries for a subexpression Q_i, at each iteration, LA chooses the "best" selection predicate, denoted by P_{opt}, such that the resultant query's result size is at least m_i and is minimized. In addition, LA further "looks ahead" other subexpressions that have not been processed and are affected by P_{opt} (i.e., the query subexpressions use the attribute in P_{opt}) to ensure that the size of the resultant query of each affected subexpression after using P_{opt} does not violate its cardinality constraint.

Implementation. Similar to the case with a single constraint, LA maintains k arrays $C_{R,1}$, $\cdots$, $C_{R,k}$ in the main memory in the same role with C_R. Each $C_{R,i}[j]$ is a boolean value to keep track of whether the j^{th} tuple of $S_{base,i}$ is still being selected after each iteration.

When considering a candidate selection predicate P_c that involves an attribute A w.r.t. $S_{base,i}$, LA sequentially scans the attribute list of A w.r.t. $S_{base,i}$. Furthermore, LA also scans the attribute lists of A w.r.t. all others $S_{base,j}$'s that have not been processed and have the attribute A in its schema to look ahead whether P_c violates the cardinality constraints on $S_{base,j}$'s.

Complexity. The running time of LA is $O(\sum_{i=1}^{k}(|S_{base,i}|))$, and the space complexity is $O(\sum_{i=1}^{k}|S_{base,i}|)$.

Example 6.3 *Consider a partial query Q on the Song database with Q_{base}:*

$$SELECT * FROM\ Song,\ Album\ WHERE\ Song.album = Album.album$$

and $C_{ans} = \{|Q_1| = 3, |Q_{base}| = 4\}$*, where* Q_1 *is "SELECT * FROM Album".*

LA *will instantiate queries for* Q_1 *first and then* Q_{base}*. To instantiate queries for* Q_1*,* LA *considers all candidate selection predicates in the form "A op v" with A is either year or sales. For sales attribute, although the selection predicate* P_{sales}*: "sales* > 75*" makes the resultant query's size minimized (i.e.,* 3*), this predicate is discarded since with the "look ahead" strategy,* LA *observes that the resultant query's size of* Q *after using* P_{sales} *is* 3*, which is less than the required constraint value (i.e.,* 4*).*

For year attribute, LA *derives "year* $\leq$ 1980*" as the optimal selection condition, which makes* Q_1 *satisfy the constraint, and the constraint in* Q *is not violated. Finally,* LA *initializes* Q_1 *into* $\sigma_{year \leq 1980}(Album \bowtie Song)$*, and derives an instantiated query for* Q_{base}*:*

$$SELECT * FROM\ Song,\ Album\ WHERE\ Song.album = Album.album\ AND$$
$$year \leq 1980\ AND\ length \leq 6.$$

□

6.3.3 Sampling-based Approach

This section introduces a variant of LA, referred to as LA^e for the estimation version of LA, that utilizes a sampling-based approach to improve the efficiency at the cost of generating approximate solutions. Our experimental results show that the results produced by LA^e to be reasonably comparable to LA.

For simplicity and without loss of generality, we consider the case with a single cardinality constraint, which requires $|Q_{base}| = m$. Recall that the main idea of LA is to initialize $Q_{inst} = Q_{base}$, and select an optimal selection predicate P_{opt} at each iteration so that $|\sigma_{P_{opt}}(Q_{inst})|$ is minimized and at least m. LA currently computes $c_{opt} = |\sigma_{P_{opt}}(Q_{inst})|$ *exactly* using a data-driven approach.

Table	Symbol	# Tuples
adult	*adult*	45222
lineitem	LI	6001215
order	O	1500000
partsupp	PS	800000
part	P	200000
customer	C	150000
supplier	S	10000
track	track	10000000

Table 6.2: Table sizes (number of tuples)

$\texttt{LA}^e$, however, saves the computation by only *estimating* the value for c_{opt}. $\texttt{LA}^e$ basically takes a random sample from S_{base}, denoted as S_{sample}, and performs P_{opt} on S_{sample} to estimate c_{opt}. $\texttt{LA}^e$ can utilize some well-known sampling-based procedures for estimating join selectivity in literature (e.g., *t_index*, *p_index* [19]) that guarantee good error bounds for estimating c_{opt}. In this work, $\texttt{LA}^e$ follows *t_index* method; the other sampling-based methods for estimating join selectivity can be applied into our framework too.

Therefore, the framework of $\texttt{LA}^e$ consists of two steps. First, $\texttt{LA}^e$ takes a sample S_{sample} of S_{base} using *t_index* procedure, and then applies the methods of LA where S_{sample} takes the role of S_{base}.

6.4 Experimental Study

In this section, we evaluate the effectiveness and the efficiency of our proposed techniques to support the two modes of processing partial queries. In Section 6.4.1, we evaluate the performance of DP and Greedy in terms of their running time and the quality of their computed results. In Section 6.4.2, we demonstrate the effectiveness of LA to support PQI, and compare the trade-offs of LA and $\texttt{LA}^e$ in terms of the running time and the quality of instantiated queries. We also compare the quality of instantiated queries returned by our methods with those returned by TQGen, the state-of-art approach proposed in [35].

Q$_1$: $Q_{base} = \pi_*(adult)$ $maximize(sum(capitalloss)) \leq 1000$ $count(occupation) = 4$	**Q$_2$**: $Q_{base} = \pi_*(adult)$ $maximize(sum(edunum)) \leq 1000$ $groupby(nativecountry, card) \leq 5$
Q$_3$: $Q_{base} = \pi_*(adult)$ $maximize(sum(capitalloss)) \leq 3000$ $3 \leq count(occupation) \leq 4$ content(occupation, "Sales")	**Q$_4$**: $Q_{base} = \pi_*(adult)$ $maximize(sum(edunum)) \leq 2000$ $count(nativecountry) = 2$ $count(race) = 2$ content(nativecountry, "US") $groupby(nativecountry, race, count, *) \leq 2$
Q$_5$: $Q_{base} = \pi_* \sigma_{length \geq 240000}(track)$ $maximize(sum(length)) \leq 30000000$ milliseconds $count(artist) = 5$ content(artist, "Bob Dylan")	**Q$_6$**: $Q_{base} = \pi_*(part)$ $maximize(sum(retailprice)) \leq 80000$ $count(brand) = 4$
Q$_7$: $\pi_*(partsupp)$	**Q$_8$**: $\pi_*(lineitem \bowtie order \bowtie customer)$
Q$_9$: $\pi_*(partsupp \bowtie part)$	**Q$_{10}$**: $\pi_*(partsupp \bowtie part \bowtie supplier)$

Table 6.3: Partial queries for experiments

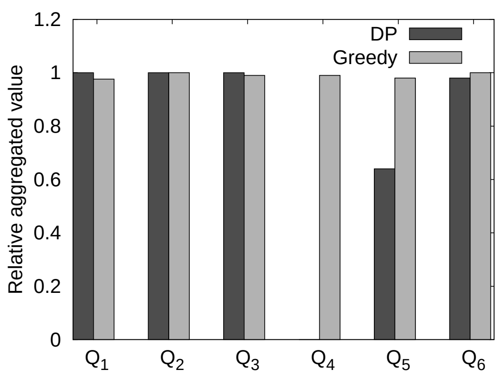
We used three real data sets for the experiments: Adult[2], TPCH (with a database size of 1GB), and a music data set containing $10,000,000$ songs and $500,000$ artists[3]. We used ten test queries including four queries on Adult data set (Q_1 to Q_4), one query on music data set (Q_5), and five queries on TPCH data set (Q_6 to Q_{10}). Queries Q_7 to Q_{10} are the test queries used in [35]. The size of the test data is shown in Table 6.2, and the test queries are shown in Table 6.3.

We used PostgreSQL 8.3 for our database system, all algorithms were coded using C++ and optimized with GNU C++ compiler. Our experiments were conducted on a dual-core, 2.33GHz PC running Linux with 3.25GB of RAM and a 250GB hard disk.

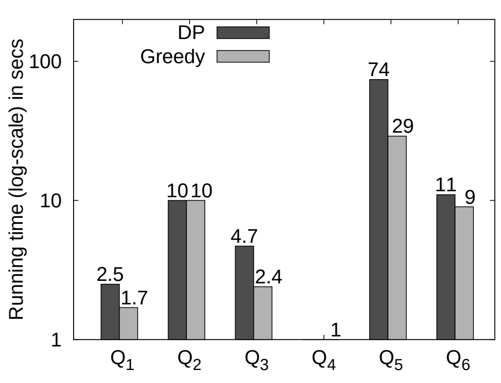
6.4.1 Evaluating Partial Queries

This section compares `DP` and `Greedy` to evaluate partial queries in terms of the quality of computed results and the running time. We used four partial queries (Q_1 to Q_4) that involve small data sets and another two partial queries (Q_5 and Q_6) that involve large data sets.

[2]http://archive.ics.uci.edu/ml/datasets/Adult
[3]http://musicbrainz.org/



(a)



(b)

Figure 6-3: Comparison between DP and Greedy

Quality of Computed Results. Figure 6-3(a) compares the quality of the results computed by DP and Greedy in terms of the relative aggregated value, defined as the ratio between the actual aggregated value returned by a method and the required maximum aggregated value. The results show that for queries Q_1 to Q_3, the aggregated values returned by DP are optimal. The aggregated values derived by Greedy are reasonably lower (i.e., 3% lower) than DP.

Query Q_4 is an example of a general partial query that contains all kinds of constraints supported in this work and DP cannot handle. Thus, the DP result for Q_4 is not shown in Figure 6-3. For this query, Greedy can find one set of tuples that satisfies all the constraints.

For queries that involve large data sets (Q_5 and Q_6), since the constructed matrices for dynamic programming are too large to fit in the main memory, DP used its approximation version to scale down the domain values of the attributes used with the aggregated constraints (e.g., length, retail price attributes). For Greedy, with the heuristic strategy, Greedy first selected a set of tuples satisfying the count constraints, thus reducing the number of tuples to be considered by the dynamic programming for the sum optimization constraints. Therefore, the solution of Greedy can be better than DP in these cases. In fact, the results of Greedy for Q_6 is slightly better than DP. For queries Q_5, with the number of involved tuples and the number of distinct values of the attribute used with the count constraint are really large (3727521 and 270352 tuples, respectively), `Greedy` returns much better quality result than `DP`. The aggregated value returned by `Greedy` is around 1.5 times larger than the ones by `DP`.

Running Time. Figure 6-3(b) compares the running time of `Greedy` and `DP` to return one result set for each query. `Greedy` runs 1.5 - 2.5 times faster than `DP`. The result is expected since `Greedy` is a heuristic solution.

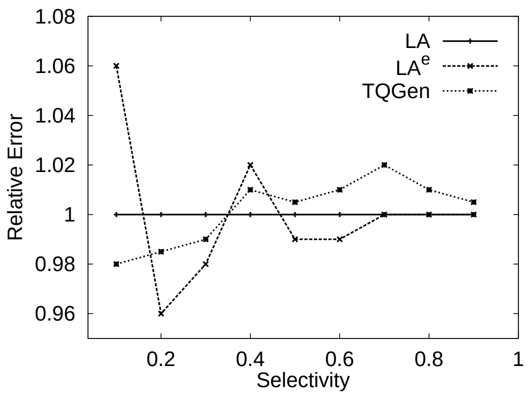
In summary, `DP` can find better quality solutions with a trade-off of slower running time compared to `Greedy` in the order of 1.5 - 2.5 times. The returned results by `Greedy` are reasonably comparable to `DP`. In addition, `Greedy` can scale better than `DP` for large data sets.

6.4.2 Instantiating Partial Queries

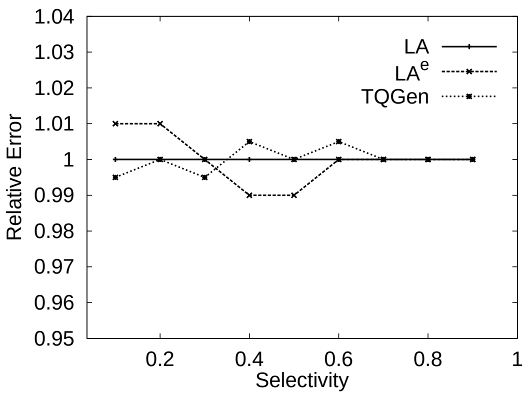
In this set of experiments, we compare the effectiveness of `LA` and $\texttt{LA}^e$ for `PQI` using queries Q_7 to Q_{10}. We also compare our methods with TQGen, the state-of-art approach proposed in [35]. We disabled the backtracking option for `LA` and $\texttt{LA}^e$ and reported the first instantiated query returned by these methods for most of our test queries except for query Q_{10} with five cardinality constraints, which will be described at the end of this section.

Comparing `LA` & $\texttt{LA}^e$. We compare these methods in terms of the quality of instantiated queries and the running time. Following [35], we use the *relative error* to compare the quality of an instantiated query, defined as $\frac{N^r}{N}$, where N is the target cardinality and N^r is the actual cardinality of the instantiated query returned by a method.

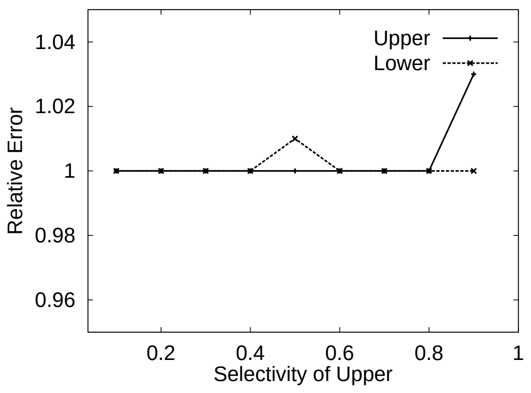
Queries Q_7 and Q_8 each has a single cardinality constraint. Q_7 is a selection query on table $PartSupp$ (PS) with the predicates on attributes *availqty* and *supplycost*. The number of tuples in the result of Q_7 is varied to be from 80K to 720K tuples; i.e. the target selectivity is from 0.1 to 0.9. Query Q_8 constraints the result of the join ($LI \bowtie O \bowtie C$) using four attributes *C.acctbal*, *O.totalprice*, *L.extendedprice*, and *L.quantity*.



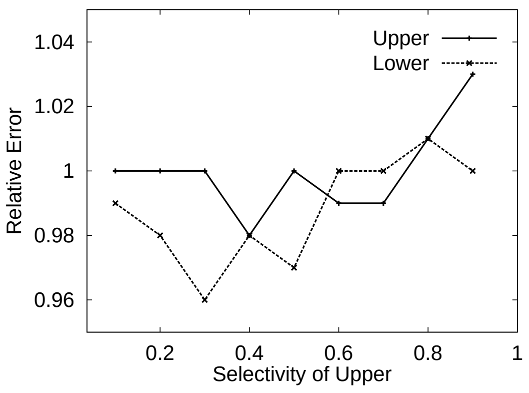
(a) *PS*



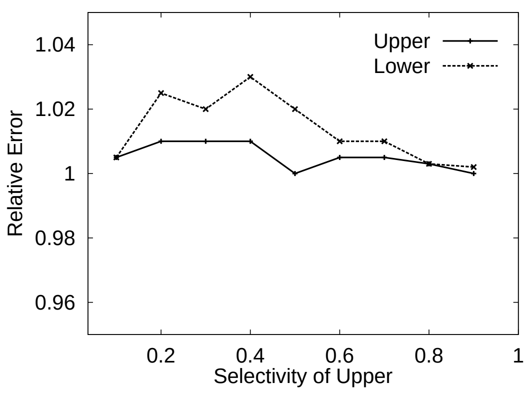
(b) $LI \bowtie O \bowtie C$



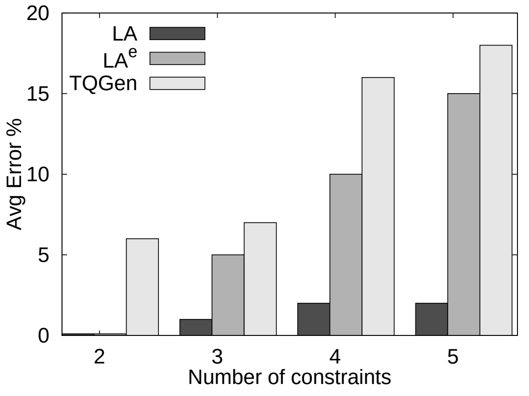
(c) $PS \bowtie P$ (LA)



(d) $PS \bowtie P$ (LA^e)



(e) $PS \bowtie P$ (TQGen)



(f) $PS \bowtie P \bowtie S$

Figure 6-4: Quality of instantiated queries

Similar to the test on Q_7, the target selectivity is also varied from 0.1 to 0.9. The results (Figures 6-4 (a)&(b)) show that while LA returned instantiated queries that precisely satisfy the cardinality constraints, the instantiated queries returned by LA^e are reasonably lower than these by LA; i.e., the relative error of instantiated queries by LA^e is consistently within 1 ± 0.06 (for Q_7) and 1 ± 0.01 (for Q_8).

Queries Q_9 and Q_{10} each has multiple cardinality constraints. Query Q_9 involves the join between PS and P; the target cardinality of PS is fixed to 700K tuples and the target selectivity of the join expression ($P \bowtie PS$) is varied between 0.1 and 0.9. The results (Figures 6-4 (c)&(d)) show that both LA and LA^e returned good instantiated queries; i.e., the errors of instantiated queries by LA and LA^e are 1 ± 0.02 and 1 ± 0.04, respectively. Note that in Figures 6-4(c),(d),(e), Upper and Lower refer to the relative error of ($PS \bowtie P$) and PS, respectively.

Query Q_{10} performs the join ($P \bowtie PS \bowtie S$) and the attributes used in the where-clause are $PS.availqty$, $PS.supplycost$, $P.retailprice$, and $S.acctbal$. The number of cardinality constraints imposed on subexpressions of Q_{10} is varied from 2 (the constraints are on P and $P \bowtie PS \bowtie S$) to 5 (the constraints are on all subexpressions of Q_{10}). We compute the average relative error over the constraints in each test case (corresponding to the selectivity from 0.1 to 0.9), and report the average of these errors over all the 9 test cases in Figure 6-4(f). The results show that the errors of LA are low and not greater than 2%; whereas the errors of LA^e are higher than those of LA and not greater than 15%.

Figure 6-5 illustrates the benefits of LA^e over LA in terms of the running times where LA^e runs 10 times faster than LA. This result is expected since LA^e manipulates much smaller data structures (in the order of 100KB) whereas LA operates on larger data structures (in the order of 100MB).

Comparing with TQGen. We compare our methods with TQGen in terms of the quality of instantiated queries. The results of TQGen in Figures 6-4(a)-(f) are extracted from [35]. We observe that the errors of TQGen are higher than those of LA and comparable with those of LA^e. For instance, the relative errors of TQGen for Q_8 are slightly

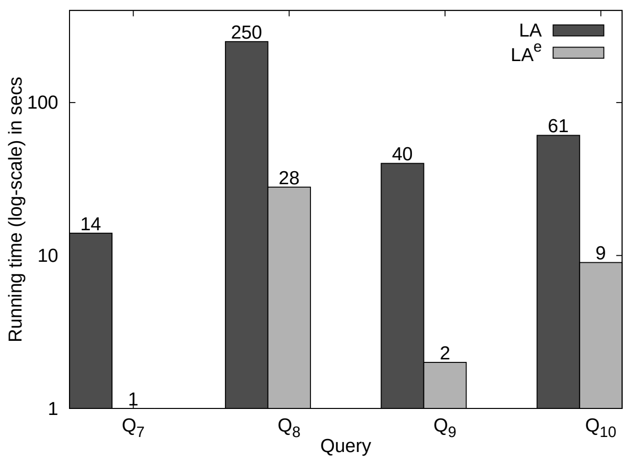


Figure 6-5: Running time comparison

lower than those of LA^e whereas the errors of TQGen for Q_{10} are slightly higher than those of LA^e.

In addition, we conduct another experiment to generate instantiated queries for Q_7 to Q_{10} without constraining attributes to be used in the selection predicates of the query. Note that TQGen cannot be applied to instantiate partial queries in these scenarios. The results in Figures 6-6(a)-(e) show that both LA and LA^e can also return high-quality instantiated queries.

The refined queries reported in Figures 6-4(a)-(f) and Figures 6-6(a)-(d) are the first one returned by LA and LA^e. For query Q_{10} with five cardinality constraints on all of its subexpression and when users do not constrain the attributes to be used the selection predicates of the instantiated queries (Figure 6-6(e)), the first refined query returned by LA and LA^e has high relative error (e.g., larger than 100%). We turned on the control knob for this case to search for the second refined query and yet obtained the good-quality refined queries by both LA and LA^e, as shown in Figure 6-6(e).

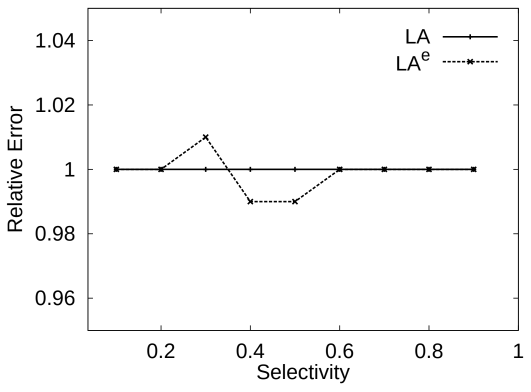
Examples of Instantiated Queries. We show some examples of instantiated queries returned by LA and LA^e in Table 6.4 for the scenarios when users restrict the attributes that can be used in the selection predicates of the instantiated queries. Here $Q_{i,s}$ (resp. $Q^e_{i,s}$) represent the instantiated queries returned by LA (resp. LA^e) for the constraint values corresponding the selectivity factor of 0.1.

	Instantiated Query
$Q_{7,s}$	$\pi_{supplycost \geq 900}\ (partsupp)$
$Q^e_{7,s}$	$\pi_{availqty \leq 985}\ (partsupp)$
$Q_{8,s}$	$\pi_{totalprice \leq 76413}\ (lineitem \bowtie order \bowtie customer)$
$Q^e_{8,s}$	$\pi_{extendedprice \leq 622087}\ (lineitem \bowtie order \bowtie customer)$
$Q_{9,s}$	$\pi_{availqty \leq 8729 \wedge retailprice \leq 1122}\ (part \bowtie partsupp)$
$Q^e_{9,s}$	$\pi_{availqty \leq 6251 \wedge retailprice \leq 1172}\ (part \bowtie partsupp)$
$Q_{10,s}$	$\pi_{supplycost \geq 900 \wedge retailprice \leq 1100 \wedge acctbal \leq 219.83}\ (part \bowtie partsupp \bowtie supplier)$
$Q^e_{10,s}$	$\pi_{retailprice \leq 1093 \wedge supplycost \leq 955 \wedge acctbal \leq 6262}\ (part \bowtie partsupp \bowtie supplier)$

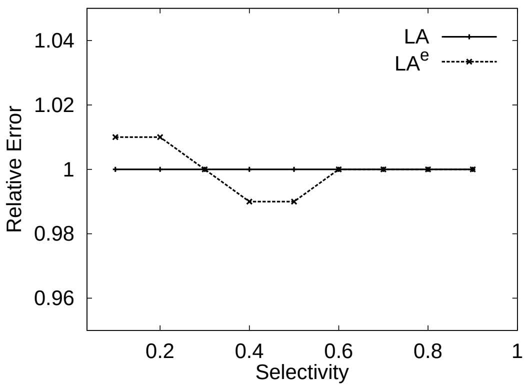
Table 6.4: Instantiated queries for Q_7 - Q_{10}

6.5 Summary

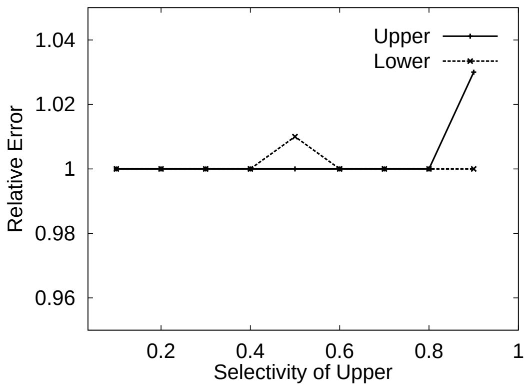
In this chapter, we introduced the concept of *partial queries*, which allows a flexible way to express a desired set of data using constraints (the third variant of QBO). In contrast to the conventional "complete" relational queries where there is exactly a set of tuples satisfying a query, a partial query could have multiple results due to the application of the constraints. We have presented two modes of processing partial queries, which are useful in different use-cases. The first evaluation mode, which evaluates a partial query to compute one or more answers, is useful for data retrieval applications. The second instantiation mode, which instantiates a partial query into one or more conventional relational queries, each of which computes an answer to the partial query, is useful for generating targeted queries in database testing. We have proposed novel algorithms for both query evaluation and instantiation, and experimentally demonstrated their effectiveness and efficiency.



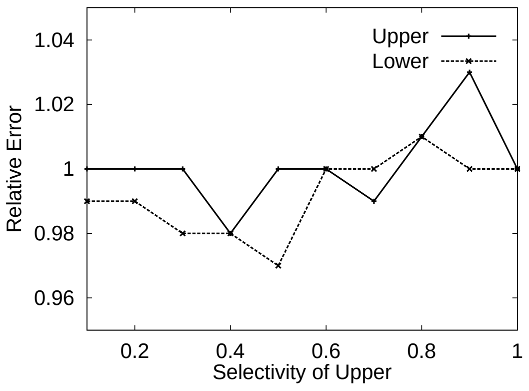
(a) PS



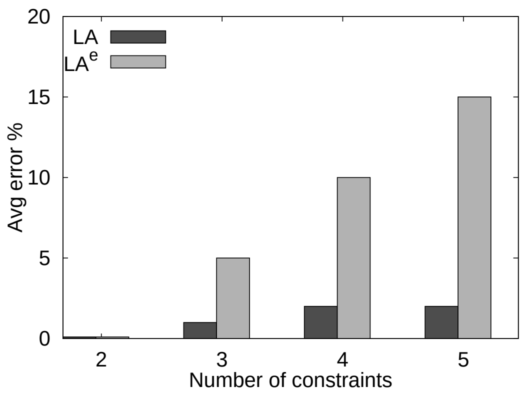
(b) $LI \bowtie O \bowtie C$



(c) $PS \bowtie P$ (LA)



(d) $PS \bowtie P$ ($\mathtt{LA}^e$)



(e) $PS \bowtie P \bowtie S$

Figure 6-6: Quality of instantiated queries (without constraining selection attributes)

Chapter 7

Conclusion

It has recently been asserted that the usability of a database is as important as its capability [23]. In this study, we have introduced a novel data-driven approach, called *Query by Output (*`QBO`*)*, that has many useful applications in not only database usability but also in other fields such as data security, data exploration & analysis, database testing, and data retrieval. In contrast to the conventional querying that takes an input query Q and computes its output, denoted by $Q(D)$, w.r.t. an input database D; the basic idea of `QBO` is to take as input the query result $Q(D)$ of some query Q, and compute a set of instance-equivalent queries $Q'_1, \cdots, Q'_n$ such that each $Q'_i(D)$ is (approximately) equal to $Q(D)$. We investigated three settings of `QBO` in this work, the main contents of these settings are summarized in Table 7.1.

QBO Problem	**Parameters**	
	Input query Q	**Given result table T**
The first variant (published in [49])	- Q is known - Q is unknown	$T = Q(D)$ T is a set of specific tuples
The second variant (published in [48])	Q is known	$T = Q(D) \bigcup S$ S is a set of tuples that are not present in $Q(D)$
The third variant (published in [50])	Q is partially specified	T is a set of constraints that must be satisfied by the query result of each derived query Q'

Table 7.1: Summary of the Three Settings of QBO

In the following, we summarize our main contributions w.r.t. each of the three variants of QBO that is considered in our work (Section 7.1). We conclude this thesis with some interesting directions that are worthy of further exploration (Section 7.2).

7.1 Contributions

Our first contribution is to introduce the problem of QBO that aims to derive instance-equivalent queries of a given input query Q w.r.t. a database D (the first variant of QBO). Such queries can shed light on hidden relationships within the data, provide useful information on the relational schema, as well as potentially summarize the original query. We have developed an efficient system, called TALOS, for QBO that models the QBO problem as a data classification task with a unique property that we term *at-least-one semantics*, which is inherent in the derivation of the IEQs. To handle data classification with this new semantics, we developed a new dynamic class labeling technique. In addition to the basic framework, we designed several optimization techniques to reduce processing overhead, and introduced a set of criteria to rank order output queries by various notions of utility. Our experimental results on several real database workloads of varying complexity highlighted the benefits of TALOS in generating interesting IEQs. We also generalized the first setting of QBO with the following three additional challenges: (1) the original query is not given as part of the input, (2) the derived queries are more expressive and go beyond the simple Select-Project-Join query fragment, and (3) there are multiple database versions. We presented a generalized approach (REQUERE) to address these issues, and demonstrated its effectiveness and efficiency with an experimental evaluation on real data sets.

Our second contribution is to introduce a new paradigm for explaining why-not questions on query results (the second variant of QBO). Our approach, named ConQueR, is based on automatically generating a refined query, whose result includes both the original query's result as well as the user-specified missing tuples. In contrast to the existing explanation models [9, 22], our approach goes beyond merely identifying the "culprit" query operator responsible for the missing tuples, and is useful for applications where it is inappropriate to modify the database to obtain missing tuples. We have proposed novel algorithms to generate good quality refined queries that are not only similar to the original query, but also produce (approximately) precise query results with a small number of irrelevant tuples. Besides the basic SPJ queries, ConQueR can also answer complex why-not questions on SPJ queries with aggregation that involve comparison constraints. Our experimental results demonstrated that ConQueR not only offers a more flexible approach to explain why-not questions, but its constraint-based method of deriving refined queries is also more efficient than the classification-based method of TALOS for the first variant of QBO.

Our third contribution is to introduce the concept of *partial queries*, which allows a flexible way to express a desired set of data using constraints (the third variant of QBO). In contrast to the conventional "complete" relational queries where there is exactly a set of tuples satisfying a query, a partial query could have multiple results due to the application of the constraints. We have presented two modes of processing partial queries, which are useful in different use-cases. The first evaluation mode, which evaluates a partial query to compute one or more answers, is useful for data retrieval applications. The second instantiation mode, which instantiates a partial query into one or more conventional relational queries, each of which computes an answer to the partial query, is useful for generating targeted queries in database testing. We have proposed novel algorithms for both query evaluation and instantiation, and experimentally demonstrated their effectiveness and efficiency.

7.2 Future Directions

There are many research venues relating to *Query by Output* problem that future studies can undertake. We discuss below some of these interesting directions.

An Alternative Hybrid Solution for QBO. For the first variant of QBO, it is also interesting to explore an alternative hybrid approach for QBO that includes an offline phase to mine for soft constraints in the database and an online phase that exploits both the database contents as well as mined constraints. It is expected that such hybrid approach can run more efficiently than TALOS. There are two challenging questions to design such hybrid approaches regarding what kinds of soft constraints to mine, and how to store and retrieve these soft constraints in combination with the data-driven approach of TALOS.

Extending TALOS. Another future work is to extend the first variant of QBO for the *incremental version* of the problem setting as follows. Consider the scenario when users slightly modify the input tables; i.e., after finding the IEQs for a given input table T, a user wants to generate the IEQs for another input table T', which differs slightly from T by adding/removing some tuples from T. The current techniques of TALOS can be applied to derive IEQs for T' *from scratch*. It is also useful to adapt TALOS for this scenario to enhance the performance of finding IEQs for T' using the computation of deriving IEQs for T, which has already been executed. This problem setting reminisces the traditional incremental decision tree updates [51]. However, with the "at-least-one" semantics introduced in QBO, it is challenging to adapt the existing techniques for the incremental decision tree building into this incremental version of QBO.

Extending ConQueR. For the second variant of QBO, our current method of ConQueR explains why-not questions w.r.t. queries in SPJ and SPJ + union/aggregation fragments. It is also valuable to support other fragments of SQL queries such as top-*k* queries. For instance, consider a query that finds top-5 favorite movies, and the result does not contain the movie "Titanic", which is one of the most favorite movie of the user. The question is then why "Titanic" is not in the list of the returned movies. Explaining such situation is non-trivial, since top-*k* queries involve ranking functions that are not handled by the existing explanation models (and ConQueR as well) for why-not questions.

Reverse-engineering Dataflow Program. A recently popular data processing paradigm is *dataflow programming*, where processing is organized in acyclic graphs. Source nodes of the graph denote the input data sets, and sink nodes represent data sets to be generated. Intermediate nodes are the set-transformation operations from a suite of operator templates [39]. Similar to programming paradigms, the process of creating a dataflow is an iterative one: user makes an initial composition of the program, executes it, and analyzes the results to make further changes. The process is repeated until the systems generate desired results. Clearly, such a manual trial-and-error process is very cumbersome and time-consuming. It is desirable to automatically or semi-automatically help users to generate correct dataflow program. We can formulate this desirable functionality as a generalization of the "reverse-engineering" aspect of QBO problem as follows. Given a template dataflow program where the input data sets are known, the output data sets are also known or given in the form of the desired properties, and the intermediate nodes are either known or selected from a set of templates. The problem is to select the "correct" template(s) at each intermediate node so that the "instantiated" dataflow program will produce the desired output. At a high level, our basic formulation of QBO is a special case of this generalized setting where the dataflow program consists of only one intermediate node (i.e., the query to be reverse-engineered). It is of challenges to support this general setting of QBO, since we have to deal with multiple intermediate operators. Furthermore, these operators can be of different types (e.g., user defined functions) in addition to the conventional SQL queries that are handled in the basic setting of QBO.

Bibliography

[1] Senjuti Basu Roy, Sihem Amer-Yahia, Ashish Chawla, Gautam Das, and Cong Yu. Constructing and exploring composite items. In *SIGMOD*, pages 843–854, 2010.

[2] Benjamin Bercovitz, Filip Kaliszan, Georgia Koutrika, Henry Liou, Aditya Parameswaran, Petros Venetis, Zahra Mohammadi Zadeh, and Hector Garcia-Molina. Social sites research through courserank. *SIGMOD Rec.*, 38(4), 2009.

[3] Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *ICDE*, pages 506–515, 2007.

[4] Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. QAGen: generating query-aware test databases. In *SIGMOD*, pages 341–352, 2007.

[5] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. *SIGMOD Rec.*, 15(2):61–71, 1986.

[6] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[7] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. on Knowl. and Data Eng.*, 18(12):1721–1725, 2006.

[8] Chee-Yong Chan, H. V. Jagadish, Kian-Lee Tan, Anthony K. H. Tung, and Zhenjie Zhang. Finding k-dominant skylines in high dimensional space. In *SIGMOD*, pages 503–514, 2006.

[9] Adriane Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.

[10] W. W. Chu and Q. Chen. A structured approach for cooperative query answering. *IEEE Trans. on Knowl. and Data Eng.*, 6(5):738–749, 1994.

[11] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms, Second Edition*. McGraw-Hill Science, 2001.

[12] Yingwei Cui and Jennifer Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, pages 471–480, 2001.

[13] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3(1):48–57, 2010.

[14] Robin Dhamankar, Yoonkyong Lee, AnHai Doan, Alon Halevy, and Pedro Domingos. iMAP: discovering complex semantic matches between database schemas. In *SIGMOD*, pages 383–394, 2004.

[15] Terry Gaasterland, Parke Godfrey, and Jack Minker. An overview of cooperative answering. *J. Intell. Inf. Syst.*, 1(2):123–157, 1992.

[16] Lise Getoor, Ben Taskar, and Daphne Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, pages 461–472, 2001.

[17] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *SIGMOD*, pages 582–592, 2001.

[18] Sudipto Guha, Dimitrios Gunopoulos, Nick Koudas, Divesh Srivastava, and Michail Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.

[19] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Arun N. Swami. Fixed-precision estimation of join selectivity. In *PODS*, pages 190–201, 1993.

[20] Melanie Herschel and Mauricio A. Hernández. Explaining missing answers to spjua queries. *PVLDB*, 3(1):185–196, 2010.

[21] Melanie Herschel, Mauricio A. Hernández, and Wang-Chiew Tan. Artemis: a system for analyzing missing answers. *PVLDB*, 2(2):1550–1553, 2009.

[22] Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1):736–747, 2008.

[23] H. V. Jagadish, Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu. Making database systems usable. In *SIGMOD*, pages 13–24, 2007.

[24] Theodore Johnson, Amit Marathe, and Tamraparni Dasu. Database exploration and bellman. *IEEE Data Eng. Bull.*, 26(3):34–39, 2003.

[25] Valentine Kabanets and Jin yi Cai. Circuit minimization problem. In *ACM Symposium on Theory of Computing (STOC)*, pages 73–79, 2000.

[26] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, Berlin, Germany, 2004.

[27] Nick Koudas, Chen Li, Anthony K. H. Tung, and Rares Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[28] Georgia Koutrika, Alkis Simitsis, and Yannis Ioannidis. Précis: The essence of a query answer. In *ICDE*, pages 69–78, 2006.

[29] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22:469–476, 1975.

[30] Maurizio Lenzerini. Data integration: A theoretical perspective. In *PODS*, pages 233–246, 2002.

[31] Eric Lo, Nick Cheng, and Wing-Kai Hon. Generating databases for query workloads. *PVLDB*, 3(1):848–859, 2010.

[32] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. SLIQ: A fast scalable classifier for data mining. In *EDBT*, pages 18–32, 1996.

[33] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Y. Halpern, Christoph Koch, Katherine F. Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *PVLDB*, 4(1):34–45, 2010.

[34] Chaitanya Mishra and Nick Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[35] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510, 2008.

[36] Amihai Motro. Intensional answers to database queries. *IEEE Trans. on Knowl. and Data Eng.*, 6(3):444–454, 1994.

[37] Inderpal Singh Mumick, Dallan Quass, and Barinderpal Singh Mumick. Maintenance of data cubes and summary tables in a warehouse. *SIGMOD Rec.*, 26(2), 1997.

[38] Ion Muslea and Thomas J. Lee. Online query relaxation via bayesian causal structures discovery. In *AAAI*, pages 831–836, 2005.

[39] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *SIGMOD*, pages 245–256, 2009.

[40] Naren Ramakrishnan, Deept Kumar, Bud Mishra, Malcolm Potts, and Richard F. Helm. Turning cartwheels: An alternating algorithm for mining redescriptions. In *KDD*, pages 266–275, 2004.

[41] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[42] Sunita Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.

[43] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.

[44] Alkis Simitsis, Georgia Koutrika, and Yannis E. Ioannidis. Generalized précis queries for logical database subset creation. In *ICDE*, pages 1382–1386, 2007.

[45] Michael Stonebraker. The design of the postgres storage system. In *VLDB*, pages 289–300, 1987.

[46] P.N. Tan, M.Steinbach, and V.Kumar. *Introduction to Data Mining*. Addison-Wesley, 2006.

[47] Wang-Chiew Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4):3–12, 2007.

[48] Quoc Trung Tran and Chee-Yong Chan. How to ConQueR Why-Not Questions. In *SIGMOD*, pages 15–26, 2010.

[49] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by Output. In *SIGMOD*, pages 535–548, 2009.

[50] Quoc Trung Tran, Chee-Yong Chan, and Guoping Wang. Evaluation of Set-based Queries with Aggregation Constraints. In *CIKM*, 2011.

[51] Paul E. Utgoff. Incremental induction of decision trees. *Mach. Learn.*, 4(2):161–186, 1989.

[52] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, 1987.

[53] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 1979.

[54] Xiaokui Xiao and Yufei Tao. Output perturbation with query relaxation. *Proc. VLDB Endow.*, 1(1):857–869, 2008.

[55] Ke Yang. Integer circuit evaluation is pspace-complete. In *Journal of Computer and System Sciences*, 2001.

[56] Xiaoxin Yin, Jiawei Han, Jiong Yang, and Philip S. Yu. Efficient classification across multiple database relations: A crossmine approach. *IEEE Trans. on Knowl. and Data Eng.*, 18:770–783, 2006.

[57] Moshé M. Zloof. Query by example. In *AFIPS NCC*, pages 431–438, 1975.

APPENDIX

A Proof of Theorem 3.1

For expository simplicity, we restate Theorem 3.1 in the following.

Theorem 3.1. Given an input query Q, we define QBO_S to be the problem to find an output query Q' w.r.t. a database D, where Q' involves only selection (with predicates in the form "A_i op c", A_i is an attribute, c is constant, and $op \in \{<, \leq, =, \neq, >, \geq\}$) such that: (1) $Q'(D) = Q(D)$, and (2) the number of operators (AND, OR and NOT) used in the selection condition is not greater than a given constant s. Then, QBO_S is believed not to be in P.

We will reduce the Minimization Circuit Size Problem (`MCSP`) to QBO_S to prove Theorem 3.1.

Definition of MCSP. The `MCSP` problem takes as inputs the truth table of a Boolean function f, a positive integer number s, and answers the question if there exists a Boolean circuit of size at most s that produces the same output as f [25]. As an example, consider an instance of `MCSP` problem that takes as inputs a truth table T consisting of four variables x_1 - x_4 (Figure A-1(a)), a number $s = 2$, and returns "yes" in this case, since there exists a circuit that is equivalent to T and has size $s = 2$ (shown in Figure A-1(d)).

Reducing MCSP to QBO_S. In the following, we describe how to reduce an instance of `MCSP` into an instance of QBO_S in polynomial time. Given the truth table T of a Boolean function f involving n binary variables $x_1, \cdots, x_n$ of `MCSP`, we create a database D containing a single relation $R(A_1, \cdots, A_n)$. Each row $(v_1, \cdots, v_n)$ of T becomes a tuple

x_1	x_2	x_3	x_4	f
1	1	1	1	1
1	1	0	1	1
0	1	1	0	1
0	1	0	0	1
0	1	1	1	0
0	1	0	1	0
1	0	1	0	0

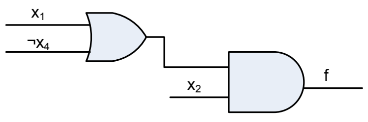
(a) Truth table T

A_1	A_2	A_3	A_4
1	1	1	1
1	1	0	1
0	1	1	0
0	1	0	0
0	1	1	1
0	1	0	1
1	0	1	0

(b) Relation R

1	1	1	1
1	1	0	1
0	1	1	0
0	1	0	0

(c) $Q(D)$



(d) The circuit

$$Q = \pi_{A_1,A_2,A_3,A_4}\sigma_{(A_1=1\ OR\ A_4=0)\ AND\ (A_2=1)}(R)$$

(e) The IEQ

Figure A-1: Example to prove Theorem 3.1

of R, where v_i belongs to the domain of A_i attribute. We formulate the given result table $Q(D)$ in QBO_S to consist of all tuples of R corresponding to rows of T that have $f = true$. The input number "s" of MCSP is transformed into constraining the maximum number of operators (AND, OR, NOT) in the selection condition of the derived IEQs of $Q(D)$ to be at most s.

Continuing with our example, the equivalent QBO_S problem of the MCSP problem described thereof consists of a database D and a given result table $Q(D)$; where D includes a single relation R (shown in Figure A-1(b)) and $Q(D)$ is shown in Figure A-1(c).

We will prove that if QBO_S returns some IEQ Q', then we can transform $sel(Q')$ into the circuit that satisfies the original MCSP problem; thus, MCSP returns "yes" in this case. In contrast, if QBO_S cannot find any IEQs, then MCSP returns "no" answer.

Case 1: QBO_S returns some IEQ Q'. We will transform $sel(Q')$ into an "equivalent" circuit C satisfying all the conditions in MCSP, where each AND (resp. OR, NOT) operator is transformed into the corresponding AND (resp. OR, NOT) gate, each selection predicate "A_i op c" is transformed into the corresponding line of x_i, $\neg x_i$, 1, or 0. In the following, we elaborate on the details of this process. Without loss of generality, assume that $sel(Q')$ is given in the form "C_1 *AND* $\cdots$ *AND* C_ℓ", where each C_i is a disjunction of some atomic predicates of the form "A_i *op c*" or its negation "NOT A_i *op c*".

The circuit C contains $(\ell - 1)$ two-input AND gates. The first AND gate has two inputs, each of which is a circuit equivalent to the clauses C_1 and C_2, respectively. The i^{th} AND gate, $i \in [2, \ell - 1]$, includes two inputs: (1) the first one is the output of the $(i-1)^{th}$ AND gate, and (2) the other is a circuit equivalent to the clause C_{i+1}.

For a clause C_i, which is a disjunction of m atomic predicates $U_1, \cdots, U_m$, we use $(m-1)$ two-input OR gates to represent C_i; the inputs to these OR gates are derived in the similar way as described with AND gates. In particular, the first OR gate has two inputs, each of which is a line corresponding to U_1 and U_2 (to be explained later). The i^{th} OR gate, $i \in [2, m-1]$, includes two inputs: (1) the first one is the output of the $(i-1)^{th}$ OR gate, and (2) the other is the line equivalent to the atomic U_{i+1} predicate. Each atomic predicate in the form "$A_i = c$" will be transformed into either (1) a single line of x_i if $c = 1$, or (2) a single line $\neg x_i$ if $c = 0$. Each atomic predicate that is evaluated to be true (e.g., $A_i \geq 0$) is transformed into a line of value 1. Correspondingly each atomic predicate that is evaluated to be false (e.g., $A_i < 0$) is transformed into a line of value 0.

Observe that the number of gates used in C is equal to the number of operators used in $sel(Q')$. In addition, when the input lines x_i obtain the values from any arbitrary row of the truth table T, the output of C is equal to the output of f's function. Thus, C is a circuit that satisfies `MCSP`, and `MCSP` returns "yes" in this case.

Example 7.1 *For the example in Figure A-1, after formulating the* `MCSP` *problem as an* QBO_S *problem,* QBO_S *derives an IEQ* Q' *of* $Q(D)$ *as:* $\pi_{A_1,A_2,A_3,A_4}\sigma_{(A_1=1\vee A_4=0)\wedge(A_2=1)}(R)$. *We derive a circuit C from* Q'*, shown in Figure A-1(d), as follows. The derived circuit C has one AND gate; its inputs include (1) a circuit that is equivalent to the clause* C_1*:* $(A_1 = 1 \vee A_4 = 0)$*, and (2) another circuit that is equivalent to the clause* C_2*:* $A_2 = 1$. *The circuit equivalent to* C_1 *includes one OR gate, the inputs of which contain* x_1 *(representing for the clause "*$A_1 = 1$*") and* $\neg x_4$ *(representing for the clause "*$A_4 = 0$*").*

□

Case 2: QBO_S cannot find any IEQs. Assume that QBO_S cannot return any IEQ Q' for $Q(D)$ whereas MCSP returns "yes" answer; i.e., there exists a circuit C satisfying MCSP. We prove that this assumption is invalid using contradiction.

More specifically, we transform the valid circuit C for MCSP into an "equivalent" selection condition S as follows. We replace every occurrence of the variable x_i by the corresponding predicate "$A_i = 1$". Similarly, we replace every occurrence of $\neg x_i$ by a predicate "$A_i = 0$". In a similar way, we transform any AND (OR, NOT) gate into the corresponding AND (OR, NOT) operator. We then formulate a query $Q' = \pi_{A_1,\cdots A_n}\sigma_S(R)$. Clearly, Q' is an IEQ of $Q(D)$, since Q' selects tuples of R corresponding to rows of T that have $f = 1$. Furthermore, the number of operators in $sel(Q')$ is equal to the size of the circuit, and thus is at most s. Therefore, Q' is an IEQ that needs to be found by QBO_S. This fact contradicts to our assumption.

In summary, we have reduced MCSP to QBO_S in polynomial time. It has been proven that MCSP is believed not to be in **P** [25]. Therefore, QBO_S is believed not to be in **P**.

B Proof of Theorem 3.2

For expository simplicity, we restate Theorem 3.2 in the following.

Theorem 3.2. Given an input query Q, we define QBO_U to be the problem to find an output query Q' w.r.t. a database D in the form $Q' = Q_1$ *union* $\cdots$ *union* Q_k, with each Q_i is an SPJ query and the select-clause of Q_i refers to only attributes of relations in Q_i, such that: (1) $Q'(D) = Q(D)$, and (2) k is not greater than a given constant n. Then QBO_U is NP-hard.

We will reduce the Set-Covering problem to QBO_U to prove Theorem 3.2. Recall that the Set-Covering problem takes the following as inputs: (1) a set of items $U = \{a_1, a_2, \cdots, a_\ell\}$, (2) a set $S = \{S_1, \cdots, S_m\}$ where each S_i is a (non-empty) subset of U and $\bigcup_{i=1}^{m}(S_i) = U$, and (3) a constant number n. The Set-Covering returns "yes" answer if there exists k subsets S_i such that the union of these subsets is equal to U and $k \leq n$; or "no" answer, otherwise.

We first construct m relations $R_i(c_i)$, where each R_i is a single-column relation containing all elements in the corresponding set S_i as its tuples. We formulate the given result table to consist of all tuples in U (derived from the query that performs a union of tuples from all S_i, $i \in [1, m]$). We will prove that the result for the Set-Covering problem depends on whether QBO_U can find an IEQ for Q or not.

Assume that QBO_U returns an SPJU-IEQ Q' of T in the form of Q_1 *union* $Q_2 \cdots$ *union* Q_k, with $k \leq n$. Since the projected attributes in Q_i only refer to the schema attributes of R_i, for simplicity and without loss of generality, we shall assume that $proj(Q_i) = c_i$ for $i \in [1, k]$. Since $\bigcup_{i=1}^{k}(R_i) \supseteq \bigcup_{i=1}^{k}(Q_i(D))$ and $\bigcup_{i=1}^{k}(Q_i(D)) = T$, it derives that $\bigcup_{i=1}^{k}(S_i) = U$. Thus, Set-Covering problem returns "yes" answer with the collection of $S_1, \cdots, S_k$ having $\bigcup_{i=1}^{k}(S_i) = U$.

In another case, assume that QBO_U does not return any SPJU-IEQs of $Q(D)$ but the Set-Covering problem return "yes" answer; i.e., there exists k subsets $S_1, \cdots, S_k$ such that $\bigcup_{i \in [1,k]}(S_i) = U$ with $k \leq n$. We formulate a new query Q'': $\bigcup_{i \in [1,k]} \pi_{R_i.c_i}(R_i)$. Clearly, Q'' is an SPJU-IEQ of T that should be returned by QBO_U. This fact contradicts to our assumption that there does not any exist any SPJ-IEQs for QBO_U.

In summary, we have reduced from the Set-Covering problem to QBO_U. Since Set-Covering problem is an NP-complete problem, QBO_U is an NP-hard problem.

C Proof of Theorem 3.3

For expository simplicity, we restate Theorem 3.3 in the following.

Theorem 3.3. Given an input query Q, we define QBO_G to be the problem to find an output query Q' w.r.t. a database D such that: (1) $Q'(D) = Q(D)$, and (2) users can specify any constraints on the clauses of Q' (e.g., the select clause of Q' can contain arbitrary arithmetic expressions or the where-clause of Q' must contain some specific

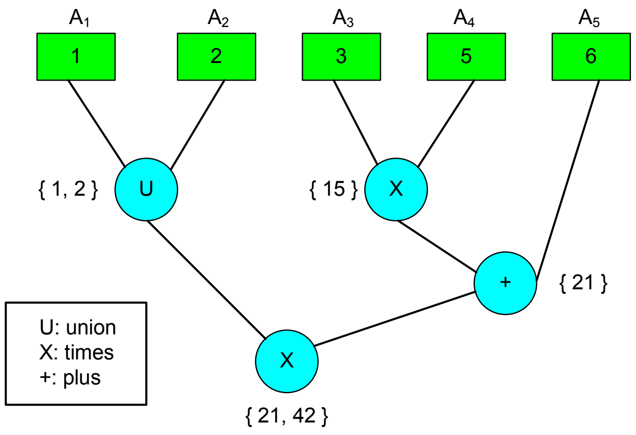


Figure C-2: An integer circuit example

selection conditions). Then QBO_G is PSPACE-hard.

We prove Theorem 3.3 by reducing the Integer Circuit Evaluation problem (ICE) [55] to QBO_G.

Definition of ICE. An integer circuit (IC) takes singleton sets as inputs, each of which contains one integer. There are three types of set operations that are considered as gates in an integer circuit: (1) the union gate, denoted by $A \bigcup B$ and defined as $A \bigcup B = \{a \mid a \in A \ \vee \ a \in B\}$; (2) the pair-wise multiplication gate, denoted as $A \times B$ and defined as $A \times B = \{a \cdot b \mid a \in A, b \in B\}$; and (3) the pair-wise addition gate, denoted as $A + B$ and defined as $A + B = \{a + b \mid a \in A, b \in B\}$. The ICE problem takes an integer X, an integer circuit C as its inputs, and determines whether X is contained in the output produced by C.

Example 7.2 *As an example of* ICE*, consider an integer circuit C shown in Figure C-2 that has five inputs* $A_1, \cdots, A_5$. *The integer circuit C calculates* $U_{12} = A_1 \bigcup A_2$, $M_{34} = A_3 \times A_4$, $P_{345} = M_{34} \times A_5$, *and produces the output* $M_{12345} = U_{12} \times P_{345}$. *Given one instance of* ICE $(C, A_1 = \{1\}, A_2 = \{2\}, A_3 = \{3\}, A_4 = \{5\}, A_5 = \{6\}, X = 21)$, ICE *returns yes, since X is in the output of the circuit.* □

Reducing ICE to QBO_G. Given an instance of ICE $= (C, X, X_1 = \{a_1\}, \cdots, X_n = \{a_n\})$, we formulate an equivalent QBO_G problem of ICE as follows. The database D of

QBO_G consisting of n tables $T_i(c_i)$ corresponding to n singleton set X_i, and another table $T_{n+1}(c_{n+1})$ consisting of one tuple (X). Each table T_i contains one row (a_i). The given result table is $Q(D) = \{(X)\}$ (derived from the input query Q: "SELECT c_{n+1} FROM T_{n+1}"). Let $\mathcal{R}$ denote the view equivalent to the given circuit C of ICE in the sense that the output of $\mathcal{R}$ on D is equivalent to the output of C on the given circuit. $\mathcal{R}$ is derived recursively as follows:

(a) If C is one of the input set X_i, then $\mathcal{R} \equiv T_i$;

(b) If C is the output of a union gate involving two inputs $\mathcal{X}_j$ and $\mathcal{X}_k$; then $\mathcal{R}$ is equivalent to $\mathcal{R}_j$ *union* $\mathcal{R}_k$;

(c) If C is the output of a multiplication gate involving two inputs $\mathcal{X}_j$ and $\mathcal{X}_k$, then $\mathcal{R}$ is equivalent to $\pi_{\mathcal{R}_j.c_j \times \mathcal{R}_k.c_k}(\mathcal{R}_j \times \mathcal{R}_k)$. Here, we use $\mathcal{R}_j \times \mathcal{R}_k$ to denote the Cartesian product between $\mathcal{R}_j$ and $\mathcal{R}_k$;

(d) If C is the output of an addition gate involving two inputs $\mathcal{X}_j$ and $\mathcal{X}_k$, then $\mathcal{R}$ is equivalent to $\pi_{\mathcal{R}_j.c_j + \mathcal{R}_k.c_k}(\mathcal{R}_j \times \mathcal{R}_k)$.

We impose a constraint on the where-clause of Q' as part of the inputs to the QBO_G problem as follows. The where-clause of Q' must be in the conjunctive form and contain the predicate: "X IN $\mathcal{R}$". Clearly, if QBO_G can return some IEQs of Q, then "X IN $\mathcal{R}$" must be true. It derives that X is accepted by the circuit C. Correspondingly, when QBO_G does not return any IEQs, then X is not accepted by ICE. It is because if X is accepted by ICE , then QBO_G must return at least one IEQ (e.g., "SELECT X FROM R WHERE X IN $\mathcal{R}$"). It has been shown that ICE is **PSPACE**-complete in [55]. Thus, QBO_G is in **PSPACE**-hard.

D Proof of the Optimality of TALOS

Let P_{m+1} denote the bound tuples in S that are labeled positive, and P_{m+2} denote the bound tuples in S that are labeled negative. We have $S = P_1 \cup \cdots \cup P_m \cup P_{m+1} \cup P_{m+2}$.

Define $n_{i,j}$ be the number of tuples in $P_i \cap S_j$, for $i \in \{m+1, m+2\}$, and $j \in \{1,2\}$. When splitting S into S_1 and S_2, there are $(n_{m+1,j} + f_j)$ tuples labeled positive and $(n_{m+2,j} + \sum_{i=1}^{m} n_{i,j} - f_j)$ tuples labeled negative in S_j. Thus, the Gini index of each set S_j, $j \in \{1,2\}$, is given by Equation 7.1.

$$Gini(S_j) = 1 - \left(\frac{n_{m+1,j} + f_j}{\sum_{i=1}^{m+2} n_{i,j}}\right)^2 - \left(\frac{n_{m+2,j} + \sum_{i=1}^{m} n_{i,j} - f_j}{\sum_{i=1}^{m+2} n_{i,j}}\right)^2 \tag{7.1}$$

The Gini index of the split S into S_1 and S_2 is computed by Equation 7.2.

$$Gini(S_1, S_2) = \alpha_1 \cdot Gini(S_1) + \alpha_2 \cdot Gini(S_2), \tag{7.2}$$

where $\alpha_j = (\sum_{i=1}^{m+2} n_{i,j})/(\sum_{k=1}^{2} \sum_{i=1}^{m+2} n_{i,k})$, $j \in \{1,2\}$. After simplifying $Gini(S_1, S_2)$, we obtain:

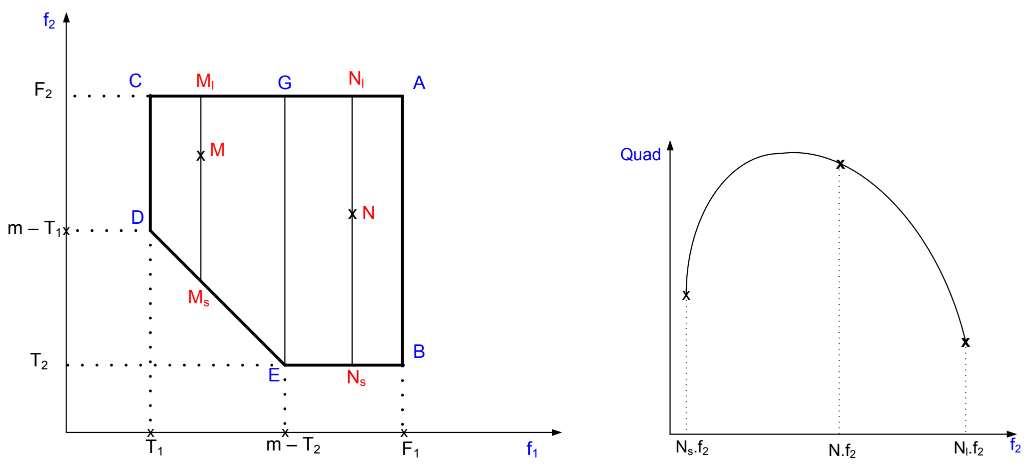
$$Gini(S_1, S_2) = c - (a_1 \cdot f_1 + b_1)^2 - (a_2 \cdot f_2 + b_2)^2, \tag{7.3}$$

where c, a_1, a_2, b_1, b_2 are constants; and f_1 and f_2 are the variables.

Let $F_j = \sum_{i=1}^{m} n_{i,j}$, $j \in \{1,2\}$. Intuitively, F_j represents the total number of free tuples in S_j. Since there are totally m sets of "types" SP_1, SP_2 and SP_{12}, the number of SP_1-sets and SP_2-sets must be no greater than m. Therefore, $T_1 + T_2 \leq m$ (A3).

Furthermore, since T_1 represents the number of SP_1-sets, it derives that $(m - T_1)$ equals to the number of SP_2-sets and SP_{12}-sets. In other words, $(m - T_1)$ is the number of subsets of free tuples, each of which has at least one tuple in S_2. Because the number of free tuples in S_2 (i.e., F_2) must be no smaller than the number of subsets of free tuples in S_2 (i.e., $m - T_1$); it derives that $F_2 \geq m - T_1$; or $T_1 + F_2 \geq m$. Similarly, we also have $T_2 + F_1 \geq m$ (A4).

Our problem to optimize the node splits of S into S_1 and S_2 becomes finding values for the two variables (f_1, f_2) such that $Gini(S_1, S_2)$ (defined in Equation 7.3) is minimized, where the values of f_1 and f_2 satisfy the following four conditions:



(a) Domain space (b) Quad graph

Figure D-3: The domain space of f_1 and f_2

(A1) $T_j \leq f_j \leq F_j$, $j \in \{1, 2\}$

(A2) $f_1 + f_2 \geq m$

(A3) $T_1 + T_2 \leq m$

(A4) $T_j + F_{3-j} \geq m$, $j \in \{1, 2\}$

The domain space of f_1 and f_2 satisfying the four conditions (A1) - (A4) is the region confined by the polygon $ABCDE$ in the two dimensional spaces of f_1 and f_2, as shown in Figure D-3(a). For a data point N in the two-dimensional space of f_1 and f_2, we use $Gini(N)$ to represent the value of $Gini(S_1, S_2)$ where f_1 and f_2 (in Equation 7.3) obtain the values of the f_1 and f_2's dimension values of the point N, respectively; i.e., $f_1 = N.f_1$ and $f_2 = N.f_2$. The optimality of `TALOS` is based on the following result.

Theorem 7.1 *The minimum value of $Gini(S_1, S_2)$, referred to as $Gini_{min}$, is the smallest value in the set $\{Gini(A), Gini(B), Gini(C), Gini(D), Gini(E)\}$.* □

We prove Theorem 7.1 by showing that $Gini_{min} \leq Gini(N)$ for an arbitrary point N locating in the polygon $ABCDE$. We consider the following two cases depending on whether N locates in the region $ABEG$ or in the region $CDEG$.

Case 1: N locates in the region $ABEG$. Let N_l and N_s be the two points at the boundary of $ABEG$ that have the same f_1 values with N; i.e., the dimensional values of N_l and N_s are: $N_l(N.f_1, F_2)$ and $N_s(N.f_1, T_2)$. Since N, N_l, and N_s have the same values on their f_1's dimension, the comparisons of $Gini(S_1, S_2)$ at the points N, N_1 and N_2 are equivalent to the comparisons of the function $\mathtt{Quad}(f_2) = c - (a_2 \cdot f_2 + b_2)^2$, where f_2 obtains the value from the set $\{N.f_2, N_l.f_2, N_s.f_2\}$. The graph of $\mathtt{Quad}(f_2)$ is shown in Figure D-3(b). Since $N_s.f_2 \leq N.f_2 \leq N_l.f_2$, it derives that $\mathtt{Quad}(N) \geq \min\{\mathtt{Quad}(N_l), \mathtt{Quad}(N_s)\}$.

Because the comparisons of $Gini(S_1, S_2)$ at the points N, N_1 and N_2 are equivalent to the comparisons of the function $\mathtt{Quad}$, we derive that $Gini(N) \geq \min\{Gini(N_l), Gini(N_s)\}$. Using the same argument, we obtain the following two inequalities: (1) $Gini(N_l) \geq \min\{Gini(A), Gini(C)\}$, and (2) $Gini(N_s) \geq \min\{Gini(B), Gini(E)\}$.

Without loss of generality, assume that $Gini(A) \leq Gini(B) \leq Gini(C) \leq Gini(D) \leq Gini(E)$. It implicitly indicates that $Gini_{min} = Gini(A)$. We have $Gini(N_l) \geq Gini(A)$, and $Gini(N_s) \geq Gini(B)$. It derives that $Gini(N) \geq Gini(A)$; in other words, $Gini(N) \geq Gini_{min}$.

Case 2: N locates in the region $CDEG$. Similar to the first case, let us consider an arbitrary point $M \in CDEG$, and two other points at the boundary of $CDEG$ that have the same f_1 values with M: $M_l(M.f_1, F_2)$ and $M_s(M.f_1, T_2)$. It is easily seen that $Gini(M) \geq \min\{Gini(M_l), Gini(M_s)\}$; where $Gini(M_l) \geq \min\{Gini(C), Gini(A)\}$, and $Gini(M_s) \geq \min\{Gini(D), Gini(E)\}$. Thus, we derive $Gini(M) \geq Gini_{min}$.

In summary, $Gini_{min} \leq Gini(N)$ for all points N in the polygon $ABCDE$. Thus, $Gini_{min}$ is the optimal value of $Gini(S_1, S_2)$.

E Proof of Theorem 6.1

For expository simplicity, we restate Theorem 6.1 in the following.

Theorem 6.1. Consider a partial query $Q = (Q_{base}, C_{ans})$ with C_{ans} containing only two count constraints: $count(A_1) = n$ and (2) $count(A_2) = m$, where A_1 and A_2 are attributes of S_{ans}. The problem of evaluating Q, referred to as PQE, is NP-complete.

It is clearly that PQE is in NP; i.e., we can verify the solution of PQE in polynomial time by checking the number of distinct values in the column A_1 and A_2 of the returned result set S_{ans}. To prove the NP-hardness of PQE, we will reduce from Set-Covering problem to PQE. Recall that the Set-Covering problem takes the following as inputs: (1) a set of items U, (2) a set $S = \{S_1, \cdots, S_n\}$ where each S_i is a (non-empty) subset of U and $\bigcup_{i=1}^{n}(S_i) = U$, and (3) a constant number ℓ. The Set-Covering returns "yes" answer if there exists k subsets S_i such that the union of these subsets is equal to U and $k \leq \ell$; or "no" answer, otherwise.

We create a relation $R(A_1, A_2)$, the tuples of which are derived as follows. For every set S_i, with $i \in [1, n]$, and each element $a \in S_i$, we insert a tuple $t = (i, a)$ into the relation R (i.e., the A_1 and A_2 values of the tuple t are i and a, respectively). Thus, there are totally $\sum_{i=1}^{n}(|S_i|)$ tuples in R with $|S_i|$ denotes the number of elements in the corresponding set S_i. In addition, there are n distinct values in A_1 column and $|U|$ distinct values in A_2 column. We set S_{base} to be equal to R and issue the following ℓ instances of PQE problem: finding a subset of S_{base} that have $count(A_1) = j$, for $j \in [1, \ell]$, and $count(A_2) = |U|$. There are two cases to consider depending on whether some instance of PQE considered above or none of them returns a solution.

Consider the case when some instance of PQE that has the iterator j equal to some value k ($\leq \ell$) returns an answer; i.e., there exists a subset $S_{ans} \subseteq S_{base}$ that have $count(A_1) = k$ and $count(A_2) = |U|$. We note that the condition $count(A_2) = |U|$ implies $\pi_{A_2}(S_{ans}) = U$; i.e., the A_2 column of S_{ans} includes all elements of U. Without loss of generality, assume the A_1 column of S_{ans} consists of k distinct values: $1, \cdots, k$. We observe that $\bigcup_{i=1}^{k}(S_i) = U$, since $\bigcup_{i=1}^{k}(S_i) \supseteq \pi_{A_2}(S_{ans})$ and $\pi_{A_2}(S_{ans}) = U$. Therefore, Set-Covering returns "yes" answer with the collection of $S_1, \cdots, S_k$ satisfies the constraint.

Consider the case when none of the considered instances of PQE above returns a solution; i.e., PQE cannot find any subset $S_{ans} \subseteq S_{base}$ that has $count(A_1) \leq \ell$ and

$count(A_2) = |U|$. We prove that Set-Covering will return "no" answer using the contradiction as follows. Assume the Set-Covering problem returns "yes", which implies that there exists m subsets such that the union of these selected sets is equal to U and $m \leq \ell$. Without loss of generality, assume these subsets are $S_1, \cdots, S_m$; i.e., $\bigcup_{i=1}^{m}(S_i) = U$. Consider a set of tuples $S_{ans} \subseteq S_{base}$ that includes all tuples of S_{base} which have $1 \leq t.A_1 \leq m$. Clearly, S_{ans} has $count(A_1) = m$ and $count(A_2) = |U|$. It derives that an instance of the PQE problem with $count(A_1) = m \leq \ell$ and $count(A_2) = |U|$ has an answer. This fact contradicts to our assumption.

In summary, we have reduced the Set-Covering to PQE. Since Set-Covering problem is an NP-complete problem, PQE is also an NP-complete problem. □

F Proof of Theorem 6.2

For expository simplicity, we restate Theorem 6.2 in the following.

Theorem 6.2. Consider a partial query $Q = (Q_{base}, C_{ans})$ with C_{ans} contains only one cardinality constraint requiring that $|Q_{base}| = n$. We define PQI to be the problem of instantiating Q into a query Q' such that the selection condition of Q' is in the conjunctive form and consists of at most ℓ predicates selected from a set of given predicates. Then, PQI is NP-hard.

We will reduce the Set-Covering problem to PQI to prove Theorem 6.2. Consider an instance of the Set-Covering problem, consisting of: (1) the universal set U, (2) m non-empty subsets $S_i \subset U$ where $\bigcup_{i=1}^{m}(S_i) = U$, and (3) a constant $\ell \leq m$. Let $S'_i = U - S_i$, $i \in [1, m]$. We construct a database D consisting of a single relation $R(A_1, \cdots, A_m)$. For each element $u \in U$, we insert a corresponding tuple t_u into R where $t_u.A_i = 1$ if $u \in S'_i$; or $t_u.A_i = 0$ if $u \notin S'_i$, for $i \in [1, m]$. We formulate an instance of PQI where Q_{base}: "SELECT * FROM R", and C_{ans} contains a constraint $|Q_{base}| = 0$. The set of given predicates consists of m predicates: $A_1 = 1, \cdots, A_m = 1$.

We will prove that if PQI returns some answer, then Set-Covering returns "yes" answer. Correspondingly, if PQI does not return any answer, then Set-Covering returns "no" answer.

Case 1: PQI returns some answer. Without loss of generality, assume the k selection predicates in $sel(Q')$ are $A_1 = 1, \cdots, A_k = 1$. It implies that $|\sigma_{(A_1=1) \wedge \cdots \wedge (A_k=1)}(Q_{base})| = 0$ with $k \leq \ell$. We will prove that $\bigcup_{i=1}^{k}(S_i) = U$ and thus Set-Covering returns "yes" answer by contradiction as follows.

Assumption:	(A1) $\bigcup_{j=1}^{k}(S_j) = U - X$ with some non-empty set $X \subset U$
Implication:	(I1) For every value $x \in X$, $x \notin S_i$ for all $i \in [1, k]$
	(I2) $x \in S'_i$ for all $i \in [1, k]$
	(I3) $\bigcap_{i=1}^{k}(S'_i) \supseteq \{x\}$
	(I4) $|\sigma_{(A_1=1)\wedge \cdots \wedge(A_k=1)}(R)| > 0$
	where (I4) contradicts to our assumption that $|Q_{base}| = 0$

Case 2: PQI does not return any answer. We will prove that Set-Covering also returns "no" answer by using contradiction as follows.

Assumption:	(A2) PQI returns no answer
	(A3) There exists k sets $S_1, \cdots, S_k$ such that $\cup_{i=1}^{k}(S_i) = U$ and $k \leq \ell$
Implication	(I5) $\cap_{i=1}^{k}(S'_i) = \emptyset$
	(I6) $Q_{base} = \sigma_{(A_1=1)\wedge\cdots\wedge(A_k=1)}(R)$ has $|Q_{base}| = 0$
	(I6) contradicts to (A2)

To derive the implication (I5), we again use the contradiction with the assumption that $| \cap_{i=1}^{k} (S'_i)| > 0$; for instance, $\cap_{i=1}^{k}(S'_i) = \{x\}$ for some value $x \in U$. It implies that x belongs to each S'_i, for every $i \in [1, k]$. Thus, x does not belong to any S_i, $i \in [1, k]$, which leads to $x \notin \cup_{i=1}^{k}(S_i)$. This fact contradicts to (A3).

In summary, we have reduced from Set-Covering problem to PQI in polynomial time. Since Set-Covering is a known NP-complete problem, PQI is an NP-hard problem.