THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

Similarity-aware Query Refinement for Data Exploration

Abdullah Mohammed Albarrak

Master of Computer Science

A thesis submitted for the degree of Doctor of Philosophy at

The University of Queensland in 2017

School of Information Technology and Electrical Engineering

Abstract

Database users are easily overwhelmed by the sheer size of data found in large-scale scientific and financial databases. Exploring these databases to make sense of the explored data and to discover interesting insights (i.e., data exploration) has been, and still is, a hideous and labour-intensive task, especially for non-expert users with no solid background of the underlying data. Some three decades ago, the database research community noticed the limitation of traditional DBMS in supporting users for data exploration tasks. Since then, the research community has proposed and designed various effective and efficient data exploration techniques to assist users in extracting interesting insights from their data. An instance of these techniques is the Query Refinement technique.

In query refinement techniques, users' queries are assumed to be imprecise, i.e., the returned result does not meet some user-defined constraints. Accordingly, the goal of query refinement techniques is to automatically refine these imprecise queries to maximize users' satisfaction with the results. In particular, the predicates of the queries are carefully modified so that the returned results satisfy the user-defined constraints. Since users' constraints on the queries results are diverse and miscellaneous, this thesis focuses on two specific forms of constraints in exploring relational and sequential data, namely, 1) user-defined aggregate constraints on the result, and 2) user-defined correlation constraints of time series data. These constraints are common in real world applications because they represent an upper level view of the result that is easier to understand and digest than the raw result itself.

This thesis addresses the limitations of current query refinement techniques that are oblivious to the similarity of the refined queries to the users' initial queries. Specifically, users' initial (and imprecise) queries are defined as anchor points for which the similarity of its corresponding refined queries are computed over the whole refinement space. Consequently, the similarity-aware query refinement problem is formulated as a search problem, which aims to balance the trade-off between minimizing the deviation from satisfying a constraint on the query result, and maximizing the similarity of the refined query to the initial one. Searching for a trade-off between satisfying a constraint on the result of a query and maximizing the similarity introduces various challenges. A common challenge shared by many query refinement problems is that finding an optimal trade-off

involves inspecting and examining a huge search space of candidate refined queries, possibly exponential. Further, evaluating candidate queries in these possibly exponential spaces to decide whether they are optimal or not incurs expensive computational and I/O costs. Hence, simply applying exhaustive solutions is not adequate since they hinder users' exploration tasks and worsen the response time. In this thesis, we discuss in detail our three key contributions, which address the challenges above in the context of query refinement for aggregate and correlation constraints.

Firstly, we formally define the Similarity-aware, Aggregate-based Query Refinement problem, in which users specify aggregate constraints on the result and prefer refined queries that are similar to their initial ones. Then, we consider the special case of aggregate constraints, in which users specify cardinality constraints on their queries results. For that special case, we propose innovative Similarity-aware Query Refinement schemes (SAQR) which employ pruning techniques to avoid unnecessary evaluations of candidate refined queries that are considered unpromising. We also show the applicability of SAQR in a web-based application (ORange) which utilizes SAQR schemes for refining selected areas based on cardinality constraints.

Secondly, we address the general case of aggregate constraints, in which multiple constraints can be defined using SQL standard aggregate operators `sum, avg, min, max`. We present EAGER schemes for this general case and propose efficient approximation and optimization techniques to elevate the shortcomings of aggregates loose bounds that are used in pruning unpromising candidate queries. Moreover, by comparison with related algorithms using real world datasets, we show the efficiency gains of our schemes under different experimental parameters.

Thirdly, we formulate the Similarity-aware, Correlation-based Query Refinement problem, in which users' queries are refined to satisfy their pairwise correlation constraints of time series data. We show the computational hardness of this problem, and propose the RELATE scheme to address the associated challenges by utilizing the incremental property of correlation. Further, we propose two-level pruning techniques for the RELATE scheme to minimize the associated computational and I/O costs. These two techniques enable RELATE to avoid exhaustively traversing the search space by pruning unqualified candidate queries, and avoid computing pairwise correlation of every time series pair wherever possible. We demonstrate by experiments the performance gains of RELATE against state-of-the-art algorithm with real and synthetic datasets.

Declaration by author

This thesis is composed of my original work, and contains no material previously published or written by another person except where due reference has been made in the text. I have clearly stated the contribution by others to jointly-authored works that I have included in my thesis.

I have clearly stated the contribution of others to my thesis as a whole, including statistical assistance, survey design, data analysis, significant technical procedures, professional editorial advice, and any other original research work used or reported in my thesis. The content of my thesis is the result of work I have carried out since the commencement of my research higher degree candidature and does not include a substantial part of work that has been submitted to qualify for the award of any other degree or diploma in any university or other tertiary institution. I have clearly stated which parts of my thesis, if any, have been submitted to qualify for another award.

I acknowledge that an electronic copy of my thesis must be lodged with the University Library and, subject to the policy and procedures of The University of Queensland, the thesis be made available for research and study in accordance with the Copyright Act 1968 unless a period of embargo has been approved by the Dean of the Graduate School.

I acknowledge that copyright of all material contained in my thesis resides with the copyright holder(s) of that material. Where appropriate I have obtained copyright permission from the copyright holder to reproduce material in this thesis.

Publications during candidature

Peer Reviewed Journal Articles

- **A. Albarrak** and M. A. Sharaf. Efficient schemes for similarity-aware refinement of aggregation queries. World Wide Web, pages 1-31, 2017.
- I. A. Ibrahim, **A. M. Albarrak**, and X. Li. Constrained recommendations for query visualizations. Knowl. Inf. Syst., 51(2):499-529, 2017.

Peer Reviewed Conference Papers

- **A. M. Albarrak** and M. A. Sharaf. Query refinement for correlation-based time series exploration. In Databases Theory and Applications - 28th Australasian Database Conference, ADC 2017, Brisbane, Australia, September 25-28, 2017, Proceedings, pages 45-58, 2017.
- **A. Albarrak**, T. Noboa, H. A. Khan, M. A. Sharaf, X. Zhou, and S. W. Sadiq. Orange: Objective-aware range query refinement. In IEEE 15th International Conference on Mobile Data Management, MDM 2014, Brisbane, Australia, July 14-18, 2014 - Volume 1, pages 333-336, 2014.
- H. A. Khan, M. A. Sharaf, and **A. Albarrak**. Divide: efficient diversification for interactive data exploration. In Conference on Scientific and Statistical Database Management, SSDBM'14, Aalborg, Denmark, June 30 - July 02, 2014, pages 15:1-15:12, 2014.
- **A. Albarrak**, M. A. Sharaf, and X. Zhou. SAQR: an efficient scheme for similarity-aware query refinement. In Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part I, pages 110-125, 2014.

Publications included in this thesis

- **A. Albarrak**, M. A. Sharaf, and X. Zhou. SAQR: an efficient scheme for similarity-aware query refinement. In Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part I, pages 110-125, 2014. Incorporated in Chapter 3, Section 3.3.

Contributor	**Statement of contribution**
Abdullah Albarrak	Conception and design (50%) Analysis and interpretation (60%) Drafting and production (60%)
Mohamed Sharaf	Conception and design (40%) Analysis and interpretation (30%) Drafting and production (40%)
Xiaofang Zhou	Conception and design (10%) Analysis and interpretation (10%)

- **A. Albarrak** and M. A. Sharaf. Efficient schemes for similarity-aware refinement of aggregation queries. World Wide Web, pages 1-31, 2017. Incorporated in Chapter 3, Section 3.4.

Contributor	**Statement of contribution**
Abdullah Albarrak	Conception and design (80%) Analysis and interpretation (90%) Drafting and production (80%)
Mohamed Sharaf	Conception and design (20%) Analysis and interpretation (10%) Drafting and production (20%)

- **A. Albarrak**, T. Noboa, H. A. Khan, M. A. Sharaf, X. Zhou, and S. W. Sadiq. Orange: Objective-aware range query refinement. In IEEE 15th International Conference on Mobile Data Management, MDM 2014, Brisbane, Australia, July 14-18, 2014 - Volume 1, pages 333-336, 2014. Incorporated in Chapter 3, Section 3.5.

Contributor	**Statement of contribution**
Abdullah Albarrak	Conception and design (70%) Analysis and interpretation (60%) Drafting and production (80%)
Tatiana Noboa	Conception and design (15%) Analysis and interpretation (10%)
Hina Khan	Analysis and interpretation (10%) Drafting and production (15%)
Mohamed Sharaf	Conception and design (10%) Analysis and interpretation (10%)
Xiaofang Zhou	Conception and design (5%) Analysis and interpretation (5%)
Shazia Sadiq	Analysis and interpretation (5%) Drafting and production (5%)

- **A. M. Albarrak** and M. A. Sharaf. Query refinement for correlation-based time series exploration. In Databases Theory and Applications - 28th Australasian Database Conference, ADC 2017, Brisbane, Australia, September 25-28, 2017, Proceedings, pages 45-58, 2017. Incorporated in Chapter 4.

Contributor	**Statement of contribution**
Abdullah Albarrak	Conception and design (80%) Analysis and interpretation (90%) Drafting and production (90%)
Mohamed Sharaf	Conception and design (20%) Analysis and interpretation (10%) Drafting and production (10%)

Contributions by others to the thesis

My principle advisor, Dr. Mohamed Sharaf, has largely contributed towards the research problems presented in this thesis. Dr. Sharaf assisted me by providing guidance and feedback on formulating the problems and solutions in this thesis. He also reviewed, polished and assisted with the published papers included as part of this thesis.

Statement of parts of the thesis submitted to qualify for the award of another degree

None.

Acknowledgements

It is with mixed feelings that I write this part of the thesis. The vicissitudes throughout my PhD journey have made me a better person, both academically and personally, for which I shall be forever grateful. Without the help and support of others, I could not have persevered and completed this journey. I would, therefore, like to sincerely thank all those who supported me along the way.

Foremost, I would like to thank my principal advisor, Dr. Mohamed Sharaf. His patience and persistent guidance with my research problems, and his compassion with my personal life situation, helped to make this journey possible. I shall always be grateful that our paths have crossed. My genuine thanks extend to my advisory committee, Prof. Shazia Sadiq and Prof. Xiaofang Zhou, for their insightful feedback and support in each step of my PhD.

I shall be forever grateful to my dear wife who stood by my side, despite being away from her parents who are in need of her. I will always be thankful for her spiritual and emotional support throughout this journey.

I would like to thank my beloved parents for their love and constant encouragement. I hope God gives me the strength to repay their love until my soul rest.

Thank you, also, to all my colleagues in the DKE group who shared my life in Australia. Special thanks to Ibrahim A. Ibrahim, Sanad Al-Maskari, Hina Khan, Saeid Hosseini and Bolong Zheng.

Lastly, I would like to formally thank my sponsor, Al-Imam Muhammad Ibn Saud Islamic University, for providing the financial support which made this journey possible.

Keywords

data exploration, query refinement, aggregate constraints, pairwise correlation constraints

Australian and New Zealand Standard Research Classifications (ANZSRC)

ANZSRC code: 080604, Database Management, 100%

Fields of Research (FoR) Classification

FoR code: 0806, Information Systems, 100%

Dedicated to . . .

my wife, Aljawharah, and my son Mohammed

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Overview

Users are easily overwhelmed by the sheer size of data in today's large-scale databases found in scientific and financial domains. Exploring these databases to make sense of the explored data and to discover interesting insights (i.e., Data Exploration tasks) has been, and still is, a hideous and labour intensive task for users [59, 52, 18, 63]. This is particularly true for non-expert users lacking a solid background of the explored database [54, 113, 81, 42, 86].

Some three decades ago [23], the database research community noticed the limitation of traditional DBMS in supporting users with their Data Exploration (DE) tasks. Since then, the community has proposed and designed various solutions and techniques to assist users with their diverse exploration tasks, with utmost effectiveness and efficiency, e.g., [25, 46, 15, 18, 127, 63, 58]. An instance of these techniques is the Query Refinement (QR) technique [17].

Given a user's query that returns unexpected results, QR techniques aim to refine this query so that its results meet a user's expectation. Specifically, the ultimate goal of QR techniques is to automatically modify (refine) predicates of a query so that the results of the modified (refined) query optimally meet the user's expectation. Achieving the goal of QR techniques entails massive computational and I/O costs, because finding an optimal refined query requires searching a huge search space of possible refined queries. Consequently, many optimization algorithms have been proposed to address the efficiency and the effectiveness aspects in achieving this goal, e.g., [78, 57, 118, 49, 83, 125, 39].

The aim of this thesis is to address the limitations of current QR techniques that are oblivious to the similarity of the refined queries to users' initial queries. In particular, we propose to define a user's initial query as an anchor point for which the similarity of its corresponding refined query is computed over the whole refinement space. Consequently, we include the similarity as an objective

when searching for a refined query to ultimately increase the user's satisfaction with the refined query. However, including this objective introduces multiple challenges (as shown in Section 1.3) and requires new, innovative algorithms with efficient optimization techniques.

The rest of this chapter is organized as follows: Section 1.2 briefly introduces the general theme of this thesis: Data Exploration. Then Section 1.3 discusses in detail the core topic of the thesis: Query Refinement. Section 1.4 discusses the key contributions of this thesis, and Section 1.5 lists the outline of the thesis.

1.2 Data Exploration

Improving the cycle of DE at different levels has recently become a major research direction for the databases research community [63]. This subsection discusses the grounds for that direction and illustrates the close links between QR and DE.

Informally, a data exploration task is a collection of ad-hoc, data-driven steps. The purpose of these steps is to make sense of the explored database and to gain interesting insights that the user would not otherwise know they exist in the first place [46, 15, 18, 127, 45]. Typically, users keep repeating these data-driven steps until they are satisfied by what they have seen, or they run out of resources (e.g., time) [15].

Unsurprisingly, traditional DBMS were not well designed for performing such DE tasks [63, 46, 128], as they were designed to provide well-structured storage for data and efficient data retrieval for well formulated and DBMS optimized queries. In [52], it is noted that this problem has been specifically identified over some three decades ago. [23] argued that DE (i.e., database usability) is not well supported by traditional DBMS because it simply was not among the concerns of the market at that time.

With recent advancements in data acquisition and storage technologies, today's databases are larger, more complex, and more difficult to explore. It is a fact that human perception of data remains constant against the exponential increase of data's volume [64], creating a large gap to be filled with efficient and effective DE techniques. Further, the parallel increase of non-experts users (e.g., journalists who want to validate politicians' claims through databases [126]), and web-based query interfaces to public and scientific databases such as the ones hosted by Google's BigQuery platform and Sloan Digital Sky Surveys (SDSS), created an urgent need for innovative DE techniques.

Accordingly, researchers have proposed highly specialized and optimized DE techniques to support users with their diverse exploration tasks. For example, some of these tasks are to recommend relevant data [30, 29], to identify interesting subspaces of data that are highly deviated from the rest of

data or a reference [124], to explain why outliers show up in the results [104, 129], to summarize and present representative sets of the potentially huge result sets [28, 65], to formulate or refine queries based on user-defined constraints [33, 119, 58, 125, 2].

1.3 Query Refinement

When querying a database, users often have some expectations of the queries results [16, 17]. For instance, a user might expects her query result to contain specific tuples [118, 49], or her query returns a non-empty result [90, 78]. When they formulate their queries and submit them to the DBMS, however, it is highly unlikely they will be satisfied with the results. That is, users oftentimes formulate wrong and imprecise queries due to their lack of a comprehensive knowledge of the data [80, 94], giving them results which do not meet their expectations.

As a result of these imprecise queries, users often enter a laborious trial-and-error process where they manually modify some predicates in their imprecise queries in the hope that these modifications will render the result to meet their expectations. In some cases, it might be impossible to perform this laborious manual process when there are usage limits on the DBMS (e.g., a maximum number of queries per session or per user). Hence, Query Refinement (QR) techniques have been proposed to address this problem by assisting users in refining their queries automatically so that the returned results meet their expectations.

Informally, QR is *"the process of refining a query when the answer to the query does not meet the expectations of the user"* [17]. In particular, the predicates of the query are automatically modified so that the refined query result reflects what the user expected, i.e., her *constraints*. A constraint over a query result is defined as the user's expectation of her query result, and it can take different forms. For example, a user might be expecting to see *a specific tuple* in the result but it was not among the returned results [118, 49]. Hence, her constraint is for the query to return a result that contains this specific tuple. Another common example is a user who expects her query to return *any* result, but the query result is empty [85, 91]. Thus, her constraint is for the query to return a non-empty result. A more common example is a user who expects her query to return a result that meets a certain *aggregate* or *correlation* value, but the returned result fails to meet that value [125, 78]. Accordingly, her constraint is for the query to return a result that meets this specific aggregate or correlation value. While these examples refer to a constraint as a singular entity (e.g., one tuple, one aggregate value), a constraint can contain multiple expectations of the same form, such as multiple tuples or multiple aggregate values.

Intuitively, exploring a query result through an upper level view representation (e.g., aggregates)

boosts users' understanding of the raw result which contains individual tuples. That is, for database users, aggregated data are easier to understand and digest than the raw data itself, and are favored in DE tasks [37, 99, 104]. Accordingly, this thesis focuses on two specific forms of constraints defined over a query result in the context of relational and time series data. Namely:

1. Aggregate Constraints (Section 1.3.1)
2. Correlation Constraints (Section 1.3.2)

Exploring the pervasive relational and time series data using the two aforementioned constraints augments users' understanding of their queries result and ultimately empowers them when exploring unfamiliar data spaces.

In the following subsections, we see that automatically refining an imprecise query to meet one of the above constraints is challenging because it requires examining a huge search space of refined queries, possibly exponential. Exhaustively examining this search space is not practical and incurs enormous CPU and I/O costs. Hence, a handful of techniques, e.g., [14, 79, 123], have been proposed to efficiently navigate this exponential space to meet users' constraints.

In light of these techniques and the two constraints mentioned above, we introduce two problems and formally address them in this thesis. Specifically, in Section 1.3.1 we show the first problem addressed by this thesis, in which users define aggregate constraints over the queries results. We show the usefulness and the applications of this problem based on a real world dataset. Then, in Section 1.3.2 we formally introduce the second problem addressed by this thesis in which users specify correlation constraints for time series pairs. Similarly, we demonstrate the applicability of this problem by an example. We also touch on the existing techniques proposed to address these two problems and their limitations.

1.3.1 Similarity-aware Aggregate-based Query Refinement

Techniques for automatically refining a query to satisfy certain aggregate constraints provide effective and efficient solutions to various problems. For instance, they can be used to address the too many/few answers problem [3, 78], to enable richer expressions in querying a database [125, 17], and to generate test queries for the purpose of database testing [79, 14].

An aggregate constraint G over a query result can be specified using an aggregate operator and an attribute, i.e., `agg(`a`)`, where `agg()` belongs to one of the standard SQL aggregation functions `count, sum, avg, min` and `max`. Further, users can define multiple aggregate constraints, i.e., $G = \{g_1, g_2, ..., g_n\}$ such that each $g_i \in G$ is a single aggregate constraint over the query result.

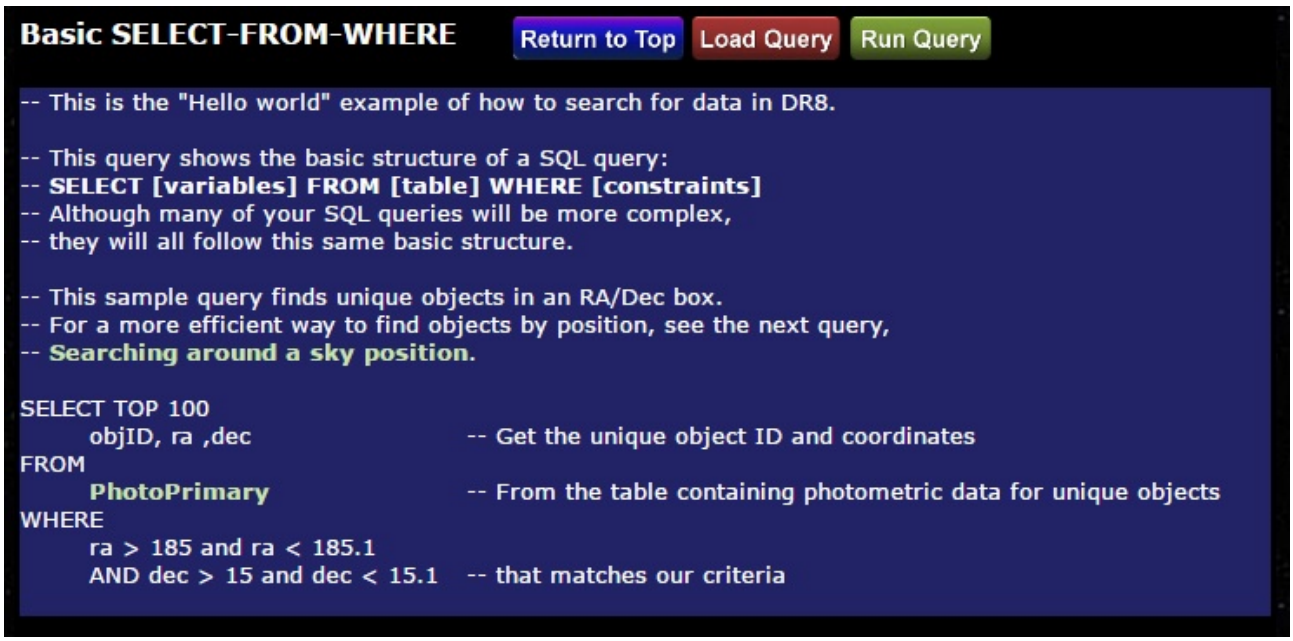

Figure 1.1: Rectangular query (or Box query) is one of the common and basic queries used to explore SDSS [112]

Refining a query is equivalent to applying modification operations on its predicates. These modification operations can be adding or removing predicates, relaxing or contracting predicates, replacing constants with other constants, joining with auxiliary tables through foreign keys, etc. As an example, a single sided range predicate on a numerical attribute a_i of this form $P_i : a_i \leq x_i$ can be relaxed (respectively, contracted) as follows: $a_i \leq x_i'$, where $x_i' > x_i$ ($x_i' < x_i$). Hence, it is easy to observe the huge number of possible refined queries (i.e., *candidate queries*) that can be generated by refining the predicates of an input query.

Let us consider the following example where specifying aggregate constraints can be effective in exploring a real-world database: the widely known Sloan Digital Sky Server (SDSS) scientific database [1]. This database is the largest map of the Universe ever made that stores details of one third of the stars and galaxies we see in the sky, and it is publicly available for anyone to explore using different interfaces, one of which is the traditional SQL query language. However, exploring this large-scale database might be an overwhelming obstacle for users, especially for those with no solid background of the database [80].

Example 1.1. *Using the SDSS database, a scientist wants to conduct a study of a particular rectangular region in the sky by retrieving astronomical objects (e.g., stars) enclosed in that region to study their properties. This type of query is one of the commonly submitted queries according to the SDSS website (Figure 1.1). We assume that the scientist has limited resources to conduct this study, e.g., time and energy, and at the same time the study has to be performed on at least 1000*

[1] http://www.sdss.org

astronomical objects to be genuine and valid.

The scientist, with her limited background of SDSS, formulates a Box query to select a rectangular region[2] *in the sky and submit it to the DBMS. Her constraint on the result is for the number of returned objects to be large enough for a valid study, yet small enough for a feasible study. Namely, her constraint is* `count(*)=1000` *objects. An example of this query is as follows:*

```
I:SELECT * FROM SDSS.PhotoPrimary
WHERE ( ra ≥ 179.5 and ra ≤ 182.3 ) AND ( dec ≥ 1.24 and dec ≤ 1.86 );
```

Since it is very difficult to precisely set the values of the `ra` *and* `dec` *predicates in query I that guarantee a desired number of objects (because of user's limited knowledge of SDSS database), the returned result might not satisfy the scientist's constraint. That is, the result might contain too few objects thereby rendering the study unreliable*[3]*, or too many objects which will make the study unachievable with the limited resources available to the scientist.*

In Example 1.1 the scientist has no choice but to iteratively try different queries and manually adjust the values for the coordinates `ra` and `dec` in her input query I, until reaching a result which satisfies her constraint. This particular special case is the cardinality-based query refinement problem, in which the aggregate constraint is equivalent to the cardinality of the result that can be computed using the `count()` aggregation function.

In [14], it was formally proven that this problem is NP-Hard. By relaxing the constraint, i.e., accepting approximate solutions that are close to the cardinality constraint, a heuristic Hill Climbing (HC) approach was developed by [14] to provide a refined query that minimizes the average relative error of the cardinality constraint in an efficient manner. Similarly, [79] addressed a similar settings of this problem where multiple cardinality constraints are defined for a query with m sub-expressions. A practical solution called TQGen has been proposed in [79] to quickly generate a refined query that optimally minimizes the sum squared logarithmic relative error of these constraints.

The interactive Semantic Windows approach [58] addresses a general case of this problem, in which users can define any aggregate constraint over a query result. We refer to this general case as the aggregate-based query refinement problem (AQR). Based on a cost-benefit model, a heuristic best-first algorithm is presented in [58] and further optimized by an adaptive prefetching technique to provide swift online results. Along the same lines, [12] presented the Package Query approach which assists users to find a set of tuples that satisfy global constraints (i.e., aggregate constraints) defined

[2] using an equatorial coordinate system

[3] A small and incorrect sample size will lead to unreliable results [6]

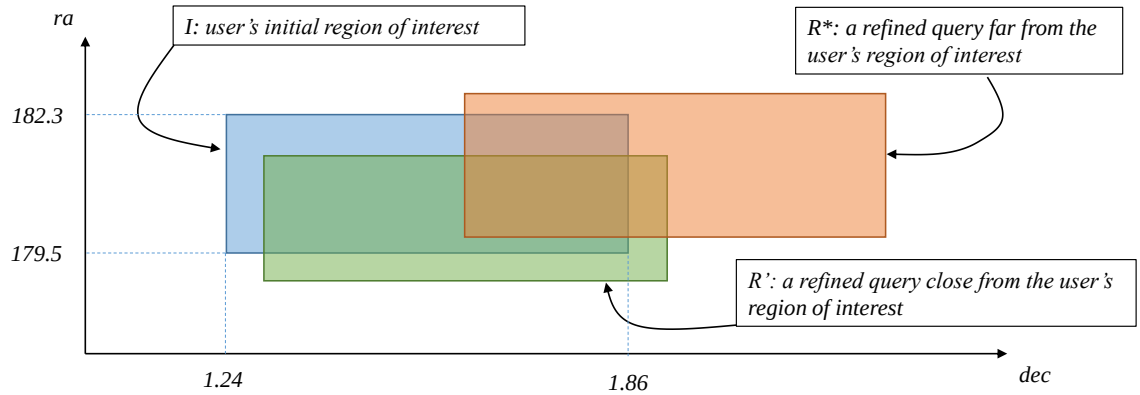

Figure 1.2: Two refined queries R^* and R' satisfy the constraint at almost the same level, but their similarities to the user's initial region of interest I are quite different

by the user. The proposed baseline algorithm in [12] translates these constraints into an Integer Linear Programming (ILP) problem and utilizes off-the-shelf ILP solvers to find a solution. A more scalable and efficient version is also proposed which relies on applying the ILP solvers on a representative set of the data to find approximated solutions, such that these solutions are guaranteed to be close by a factor to the baseline solutions.

Although the above techniques can be applied to solve the AQR problem efficiently, they exhibit various limitations. Foremost, similarity to users' input queries is completely neglected in the techniques proposed in [14, 58, 12, 79]. Going back to Example 1.1, applying any of the above techniques will help the scientist to achieve her goal, i.e., an optimal refined query R^* which optimally satisfies the cardinality constraint. However, there is no consideration given to the user's preference expressed in her input query I. That is, while R^* optimally satisfies her constraint, it might be very far from her initial region of interest, i.e., query I. At the same time, as shown in Figure 1.2 above, there might be another refined query R' that also satisfies her constraint (almost at the same level as R^*) but is very close to her initial region of interest, i.e., input query I. Therefore, suggesting R' instead of R^* as the refined query increases the users' satisfaction with the results because R' is more similar to I than R^*.

Extending these techniques to include similarity is not practicable since it introduces multiple challenges. For instance, the HC approach in [14] can be modified to choose a refinement step that reduces the relative error and dissimilarity to the input query at the same time. However, this straightforward modification will render HC vulnerable to floundering about a local minima. The Semantic Windows approach in [58] can also be modified to include similarity to an input user's query. Specifically, the dissimilarity of a cell to a query can be included in the cost-benefit model. Nonetheless, [58] represents the search space as one flat grid with a fixed granularity. Hence, the

number of cells at this approximated search space might be extremely huge when using a small value for granularity. Adding these cells to the queue and examining them exhaustively (as SW does) incurs a lot of I/O and CPU costs.

The techniques proposed in [78, 123] address the AQR problem and provide refined queries close from users' input queries, but they also have their own limitations. For instance, the SnS framework [78] provides a manual algorithm for refining a query. It is manual in the sense that users are iteratively asked to select a predicate value that is most preferred by them, hence capturing similarity manually. Other approaches such as the ACQUIRE and SAUNA [123, 57] include similarity to users' input queries in the refinement process and provide fully automatic algorithms. However, they ignore the I/O cost involved in refining a query and focus on the computational costs. For example, ACQUIRE focuses only on efficiently computing the aggregate value for a candidate query by utilizing the additive property of the aggregate. It also shares the limitation found in the SW approach: the search space is represented as one flat grid by dividing each dimension into a fixed number of partitions, resulting in a huge number of candidate queries that are exhaustively probed during the refinement process. Similarly, SAUNA requires each candidate query to be probed in order to compute its dissimilarity to the input query.

In Chapter 3 we formally address the inclusion of similarity to users' input queries in the AQR problem. Including similarity requires new methods to represent the search space, and new pruning techniques to efficiently navigate this space and find the optimal refined query. For that, we propose a suite of algorithms with optimization techniques to overcome the efficiency challenges involved in finding an optimal query. We also compare our algorithms with the HC and TQGen approaches to validate our algorithms efficiency and effectiveness.

Next, we introduce the second type of constraints which users can define to explore time series data.

1.3.2 Similarity-aware Correlation-based Query Refinement

A common thread of data exploration is querying sequences of values (e.g., time series data) to perform various tasks [97, 76, 72]. For instance, a user can query sub-intervals of time series data then compute the pairwise correlation of all pairs of time series to find correlated pairs [70, 71, 89, 100, 39] or detect patterns and anomalies [87, 102].

In some cases, users select time series data within a specific time sub-interval to compute the pairwise correlation values for all time series pairs, e.g., [70, 71, 89]. We propose to define the pairwise correlation values as *pairwise correlation constraints*, such that a user's query has to satisfy

	timestamp														
	6:00	**7:00**	**8:00**	**9:00**	**10:00**	**11:00**	**12:00**	**13:00**	**14:00**	**15:00**	**16:00**	**17:00**	**18:00**	**19:00**	**20:00**
$\mathbf{T_1}$	0.2	0.3	0.3	0.5	0.8	0.9	0.8	0.9	0.7	0.8	0.4	0.3	0.3	0.2	0.1
$\mathbf{T_2}$	0.15	0.2	0.15	0.22	0.41	0.54	0.48	0.49	0.36	0.43	0.19	0.17	0.16	0.11	0.06
$\mathbf{T_3}$	0.05	0.1	0.15	0.28	0.21	0.13	0.32	0.41	0.34	0.37	0.21	0.13	0.14	0.09	0.04

Figure 1.3: Relation R stores hourly CPU load readings of three connected servers T_1, T_2 and T_3 in a hypothetical data centre

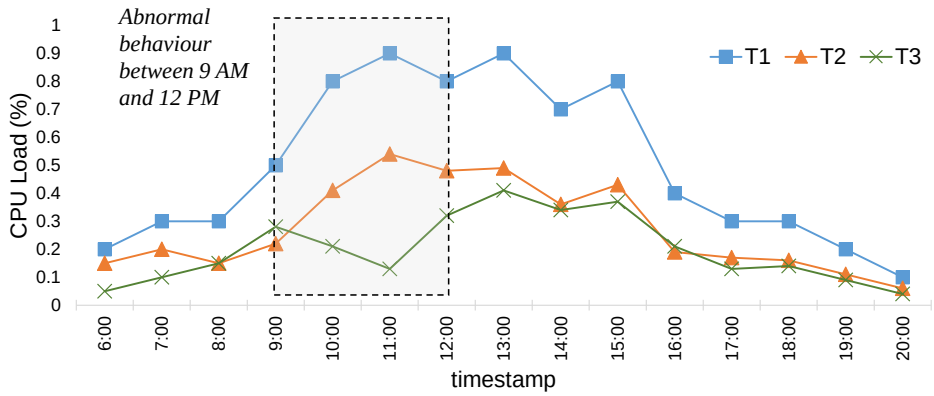

Figure 1.4: 2-D visualization of R. The abnormal behaviour is between 9 AM and 12 PM: T_1's load increases but T_2's or T_3's load does not increase (i.e., correlation close from zero)

these constraints. Specifically, in the Correlation-based Query Refinement problem (CQR), a user's input query is refined (i.e., its sub-interval is refined) such that the result of the refined query satisfies user-supplied pairwise correlation constraints for all time series pairs.

The following example illustrates how the CQR problem can be of use for users to automatically refine their queries to satisfy correlation constraints. This example is fairly prevalent in data centre management systems [73] where users analyze servers loads collectively (e.g., Queries 3 and 4 in [102]) using the correlation coefficient [114, 87, 102]. Moreover, the example will help in understanding the associated challenges for achieving the CQR problem goal.

Example 1.2. *Assume a hypothetical data centre with three connected servers T_1, T_2, T_3, where T_1 is responsible for forwarding incoming requests to T_2 and T_3 as evenly as possible. The hourly CPU load readings of these servers are stored in a database relation R, as shown in Figure 1.3.*

An admin wants to analyze the loads of these servers to identify any abnormal behaviour based on the pairwise correlation of the servers loads. Let this abnormal behaviour be: T_1's load increases but T_2's or T_3's load simultaneously does not increase (i.e., negative correlation). Conversely, the normal behaviour of these servers is for T_2 and T_3 loads to increase simultaneously as T_1's load increases (i.e., positive correlation).

Typically, a 2-D visualization of time series data is utilized for swift insights [133], such as the

one shown in Figure 1.4. While the process of visually identifying abnormal behaviours is somehow easy in this toy example as there are three time series in R, it should be clear that this process becomes more challenging when there are more series.

Consequently, the admin executes a selection query over R and then computes the pairwise correlation of the pairs (T_1, T_2), (T_1, T_3) *and* (T_2, T_3) *from the query output. These pairs and their correlation values are collectively called a correlation matrix M.*

Let the admin's initial query be:

```
Q1: SELECT * FROM R
WHERE timestamp ≤ 20 and timestamp ≥ 6;
```

The correlation matrix M_{Q_1} *for* Q_1 *is shown in Figure 1.5.*

It appears that no abnormal behaviour exists within the results of Q_1*: the loads of* T_2 *and* T_3 *follow the same pattern as their parent* T_1*, which the high pairwise correlation values in* M_{Q_1} *confirm. More precisely, the normalized sum of absolute difference (*L_1*-norm) between* M_{Q_1} *and* M_{abn} *is high (≈0.96), where* M_{abn} *represents the abnormal behaviour in pairwise correlation values, as illustrated in Figure 1.5.*

Nonetheless, between 9 and 12 there is evidence of an abnormal behaviour: T_3*'s load breaks the pattern and decreases while* T_1*'s load increases. This abnormal pattern is captured by the following query:*

```
Q2: SELECT * FROM R
WHERE timestamp ≤ 12 and timestamp ≥ 9;
```

Its matrix M_{Q_2} *is shown in Figure 1.5 as well. Figure 1.6 confirms that* Q_2*'s correlation matrix is the closest to the user-defined abnormal behaviour, i.e., among all queries,* M_{Q_2} *has the minimum sum of absolute difference to* M_{abn} *≈0.74 .*

In Example 1.2, it is assumed that the abnormal behaviour is well-known by the admin, e.g., pairwise correlation values close from -1 indicate an abnormal behaviour for the servers, as shown in M_{abn}. However, the time sub-interval (i.e., Q_2) that is the closest to this abnormal behaviour is not known to the admin. Further, it is assumed that Q_1 in Example 1.2 represents the admin's initial guess of where that abnormal behaviour is located.

To find Q_2 though, the admin is required to manually refine her input query (i.e., Q_1) by modifying the `timestamp` predicate, then compute the pairwise correlation values (i.e., correlation matrix) from

	T_1	T_2	T_3
T_1		1	0.8
T_2			0.7
T_3			

M_{Q1}

	T_1	T_2	T_3
T_1		0.98	-0.55
T_2			-0.47
T_3			

M_{Q2}

	T_1	T_2	T_3
T_1		-1	-1
T_2			-1
T_3			

M_{abn}

Figure 1.5: Correlation matrix of Q_1, Q_2 and M_{abn}. M_{abn} represents an abnormal behaviour where T_1, T_2 and T_3 have negative correlation

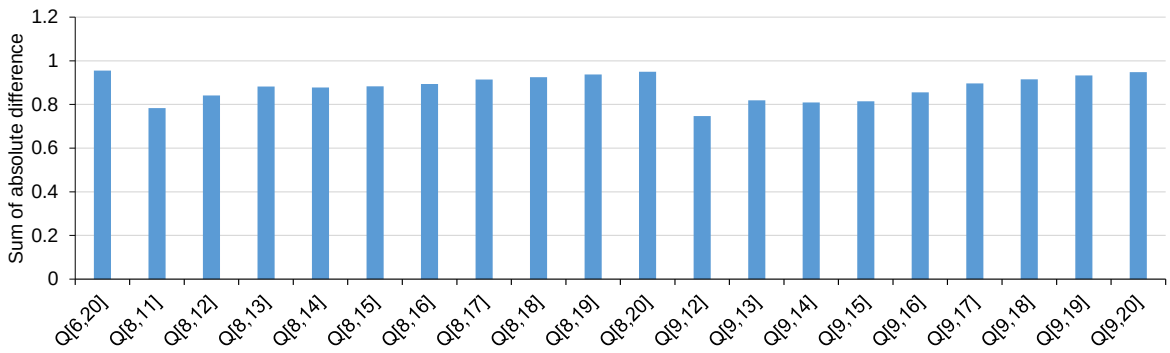

Figure 1.6: A subset of candidate queries and their normalized sum of absolute difference from the abnormal behaviour M_{abn}. $Q_2 = Q[9,12]$ has the minimum difference to the abnormal behaviour

the results of all possible refined candidate queries.

Manually examining all possible candidate queries and computing the correlation matrices out of their results to find Q_2 is obviously not a practical solution. There is a total of $\frac{m(m-1)}{2}$ candidate queries, and this number increases quadratically with the length of the time series m. Manually examining these queries is a labour intensive task and incurs tremendous I/O and CPU costs. For instance, it takes almost two hours to compute a single matrix for 10k time series [87] using a traditional PC.

A more suitable alternative is for the user to specify the target correlation matrix (e.g., M_{abn} in Example 1.2) that represents an abnormal behaviour, and an efficient solution automatically finds the query (i.e., sub-interval) that is the closest to the target. This partially resembles the Query Reverse Engineering problem [119, 120, 115], but the pairwise correlation constraints and the time series data introduce unique challenges which cannot be addressed by the techniques proposed there. For instance, [119] requires that the query result must not contain an arithmetic expression such

as correlation, whereas [115] optimization techniques are based on series of rules using aggregates properties that are not applicable to correlation.

There is a large body of work [70, 107, 87, 39] which focuses on efficiently reporting time series pairs that are above a certain correlation threshold. This body of work is tightly related to CQR problem because reporting correlated pairs requires computing correlation efficiently, which is also a core requirement in the CQR problem. For example, the techniques in [70, 71] build on the observation of [107] where correlation can be computed incrementally to report the longest sub-interval of correlated pairs. These techniques however are a special case of CQR, because there is only one correlation constraint for all time series pairs. Moreover, these techniques aim to report all time series that are correlated with a given time series, i.e., bi-variant analysis, whereas the CQR problem falls into the multi-variant analysis category, in which the correlation of all pairs of time series are reported and compared [56, 41].

In the multi-variant analysis category, the AEGIS framework [39] has been proposed for fast computation of correlation in a distributed environment. This framework goal is to report all pairs of all time series that are correlated above a certain correlation constraint, i.e., AEGIS assumes only one correlation constraint for all pairs. Further, AEGIS assumes the length of the sub-interval is known in advance, e.g., given by a user. Based on these two assumptions, the framework partitions the time series such that the pairs that are potentially correlated are contained in a single partition, to control the communication cost in a distributed environment. Extending this framework to address the general case in CQR where neither of the two assumptions are enforced is not applicable.

A similar work is presented in [87] for computing pairwise correlation of a very large number of series to automatically discover anomalies. However, it also enforces the same assumptions as in [39]: a single correlation constraint for all pairs and the sub-interval is known in advance as well.

A more recent work [128] proposes a general framework to accelerate the computation of a number of statistics such as correlation. This framework utilizes the natural overlap in users' exploratory queries to speedup the computation of statistics by synthesizing cached statistics that were previously computed. This is quite similar to the incremental computation of correlation method proposed in [107].

In Chapter 4, we address the Similarity-aware, Correlation-based Query Refinement problem, where users can specify similarity and correlation constraints. Given these constraints, we propose efficient algorithms and pruning techniques that overcome the efficiency challenges accompanied in solving this problem. We also compare the results of our algorithms to state-of-the-art algorithm using real and synthetic datasets and discuss the results under different parameters settings.

1.3.3 Challenges

Examples 1.1 and 1.2 briefly introduce the challenges related to the Similarity-aware, Aggregate-based and Similarity-aware, Correlation-based Query Refinement problems. Challenges in these problems can be classified as efficiency challenges, or effectiveness challenges.

Efficiency challenges in these two problems originate from navigating a huge search space to find an optimal refined query that satisfies the user-defined constraints. This is a common challenge shared by many data exploration problems where finding an optimal solution involves inspecting and examining a huge search space, possibly exponential, in a brute-force approach. Further, evaluating candidate solutions in these exponential spaces to decide whether they are optimal or not, incurs expensive computational and I/O costs. Hence, simply applying exhaustive solutions is not adequate since they hinder users' exploration tasks and lengthen the response time. Accordingly, the efficiency challenges of these two problems necessitate the design of innovative algorithms that can utilize properties of the constraints and the search space to provide efficient solutions. For instance, the monotonic property of aggregate operators enables an algorithm to prune unqualified queries and avoid evaluating them using aggregate bounds that are computed from previously evaluated candidate queries. Moreover, the similarity constraint enables an algorithm to early abandon evaluating a candidate query, leading to signification savings of I/O costs.

For some tasks (e.g., algorithms that translate from English to Chinese) humans can easily decide which algorithm produced the most effective output. However, modeling the process of evaluating the effectiveness of such algorithms is quite challenging. In particular, the effectiveness challenges in the two problems addressed in this thesis relate to measuring the similarity of a solution to the user's input query and user's satisfaction of the refined query. That is, how to model users' preferences so that an algorithm can automatically consider the dissimilarity of a candidate query to the input query in the refinement process. One extreme approach is to fully incorporate the user in the refinement process by asking her to label each candidate refined query based on her satisfaction of the refined query result. Although this can guarantee maximum user satisfaction with the end result, it adds a whole new substantial burden on users which can be avoided without sacrificing the solution effectiveness. Another approach which does not require users' feedback is to automatically infer the similarity of a candidate query. This can be done systematically by comparing the results of a candidate query to the input query results. However, this approach implies retrieving the results of each candidate query, which could entail high CPU and I/O costs since the number of candidate queries can be huge.

Chapters 3 and 4 present our proposed innovative techniques and algorithms which address the efficiency and effectiveness challenges involved in achieving the goals of the Similarity-aware,

Aggregate-based and Similarity-aware, Correlation-based Query Refinement problems. The proposed algorithms utilize specific properties of the constraints to prune unqualified candidate queries and employ various techniques to optimize the search process. Further, these algorithms use simple, automatic and effective methods to swiftly infer the similarity of a candidate query while searching for the optimal solution.

1.4 Contributions

Motivated by the efficiency and effectiveness challenges mentioned above that are manifested in the exploration of relational and sequential data, we have proposed a suite of optimized schemes and algorithms which guide users in refining their imprecise queries based on aggregate and correlation constraints. Specifically, this thesis makes the following key contributions:

- For a special case of aggregate constraints, i.e., cardinality of the answer, Chapter 3, Section 3.3 proposes the SAQR schemes. SAQR schemes partition the search space using a space-based partitioning method to transform it into a multi-level grid with equal-width, non-overlapping cells. Each intersection in this grid represents a candidate refined query, and one of these candidates optimally minimizes the relative error in terms of the cardinality constraint and the dissimilarity to the input query. Hence, SAQR follows a Top-K approach but at the query-level, not the tuple-level, to find this optimal candidate query, with respect of the new partitioned search space. In a nutshell, SAQR schemes utilize similarity-based and cardinality-based properties to prune and avoid evaluating unqualified candidates when searching for the optimally refined query, without any approximation.

- The special case of cardinality was then extended to allow the constraints to be any multiple aggregate constraints, i.e., `count, sum, avg, min, max` for which the EAGER scheme has been proposed with its approximation techniques in Chapter 3, Section 3.4. EAGER schemes address the limitation of loose bounds for some aggregates, the case of having multiple constraints in one query, and reduce the cost incurred in the search by strategically materializing parts of the search space.

- The applicability of SAQR schemes are demonstrated by a web-based application (ORange) presented in Chapter 3, Section 3.5. ORange was designed as a tool to efficiently guide planners in allocating services zones that optimally satisfy a certain cardinality constraint.

- Next, the Similarity-aware, Correlation-based Query Refinement problem is introduced in

Chapter 4, Section 4.2.1 which generalizes on previous problems addressed in literature in two ways: by specifying a correlation constraint for each pair of time series, and by inclusion of similarity to users' input queries in refinement.

- Then in Chapter 4, Section 4.3 the RELATE schemes are introduced as efficient solutions to this refinement problem. RELATE schemes extend state-of-the-art algorithms to incrementally compute the correlation for all pairs of time series. To achieve that incremental computation, RELATE applies two classical tree traversal methods, BFS and DFS, to visit the candidate queries in a specific order that enables incremental computation of correlation.

- Further, RELATE optimizes the search process by pruning unqualified candidate queries via monotonic properties at two levels, query similarity level, and pairwise correlation level, as explained in Chapter 4, Section 4.3.5.

1.5 Thesis Outline

This thesis is organized as follows: in Chapter 2 we introduce the preliminaries of query refinement and elaborate more on the related work. Chapter 3 presents our first contribution for the Similarity-aware, Aggregate-based Query Refinement problem. Specifically, Section 3.3 provides innovative schemes called SAQR to efficiently refine queries based on a special case of aggregate constraints, i.e., cardinality constraints on the result. Then, in Section 3.4, SAQR Schemes are extended to address the general case of the Similarity-aware, Aggregate-based Query Refinement problem where SQL standard aggregate operators `sum, avg, min, max` can be defined as constraints, and we propose efficient approximation and optimization techniques and compare them to related algorithms. Section 3.5 presents a web-based application *ORange* which employs SAQR schemes for refining selected areas based on cardinality constraints.

In Chapter 4, we formulate the Similarity-aware, Correlation-based Query Refinement problem and address its computational hardness by proposing the RELATE schemes. Section 4.3 shows the optimization techniques for RELATE schemes which include incremental computations of correlation, pruning candidate queries based on similarity constraints and pairwise correlation pruning. Finally, Chapter 5 concludes this thesis and provides suggestions for future studies in the area of query refinement.

CHAPTER 2

Literature Review

This chapter starts with a broad overview of the data exploration (DE) techniques discussed in recent literature, which were proposed to facilitate efficient and effective knowledge extraction. Then, in Section 2.2 it narrows down the discussion to one area of these techniques, Query Refinement (QR) techniques, which is the scope of this thesis.

Section 2.2 broadly divides the discussion on QR techniques based on the refinement constraints. Initially, Section 2.2.1 reviews several QR techniques that address various refinement constraints. Then, Sections 2.2.2 and 2.2.3 explore in detail the techniques proposed to address similar constraints and problems to those in this thesis: Similarity-aware, Aggregate-based Query Refinement and Similarity-aware, Correlation-based Query Refinement problems. We investigate the shortcomings of these techniques in terms of efficiency and effectiveness, and their limitations when including similarity in the refinement process.

2.1 Data Exploration Techniques

DE is central for data-driven applications in which users interact and explore data through a sequence of related queries to gain deep insights [15, 18, 127]. This new form of interaction has resulted in the design of various DE techniques capable of guiding users through the data space with utmost efficiency and effectiveness [46].

Because *"one size does not fit all"*, these techniques are application-oriented. That is, they are highly specialized and optimized for certain data exploration *objectives*. For example, some of these objectives are to recommend relevant data [30, 29], to identify interesting subspaces of data that are highly deviated from the rest of data or a reference [124], to explain why outliers show up in the results [104, 129], to summarize and present representative sets of the potentially huge result sets [28, 65], and to formulate or refine queries based on user-defined constraints [33, 119, 58, 125, 2].

Although these techniques have different objectives, they share commonality in their assumptions and optimization methods. One common assumption is that users are unfamiliar with the data space and aim to efficiently extract deep and interesting insights from their data with no prior assumptions. There are several optimization methods that are common among these techniques as well, such as incremental computations, materialization, sharing of computations, pruning based on properties and bounds, caching, etc. Next, we expatiate on these techniques then dwell on Query Refinement techniques as this is the scope of this thesis.

Data Recommendation: Recommending relevant data based on users' queries helps users to understand their queries better. The YMALDB framework [30] and the AIMQ approach [92, 93], for instance, allow users to discover highly correlated and similar tuples to the original query's results, although these discovered tuples are not among the original query's results. At the other extreme, SeeDB [124] presents and recommends alternative queries (i.e., views) that are highly deviated from a reference query's result. Such alternative queries are considered interesting and insightful for users.

Explaining Outliers in Queries' Results: Explaining outliers in queries results is another application-oriented DE technique that aims to give meaningful explanations which cause outliers to appear in the results. These explanations can be a set of tuples that must be removed from the result or modifications to the original query's predicates [129, 104].

Query Results Summarization: Summarizing queries' results assists users when exploring large-scale data, such as scientific and financial data. Specifically, because users' exploratory queries will most likely return large results, deriving insights from these large results becomes an overwhelming obstacle. Techniques which summarize and select small representatives out of the raw and large results elevate this obstacle and efficiently enhance users understanding of their queries' results. Two well-known techniques which follow this direction are the traditional Skyline [9] and Top-K [47] techniques. Other recent emerging techniques such as Regret Minimization [95, 96] and result Diversification [28, 65] have also been shown to be effective in promoting users' understanding of their queries' results. Another body of work [55] enhances the well-known drill-down operation in OLAP by showing rules along side the output. These rules represent the interesting aspects of the explored data.

Query Formulation: Query formulation techniques are orthogonal to QR techniques. Their goal is to efficiently and effectively guide users in locating their interests within a large data space by formulating a query *from scratch*, i.e., users do not provide input queries as traditionally assumed in QR techniques. That goal can be achieved by following different approaches. For example, in some cases users provide a set of tuples (i.e., result of an arbitrary query) for which they want a set of queries that return these exact tuples [119]. Discovering these queries increases users' understanding

No	**Constraint**	**Informal Definition**	**Related Works**
1	Query's result size $\geq K$, $\leq K$, or > 0	Given that Q returns too-few, too-many or an empty result, minimally refine Q to Q' to satisfy the constraint	[90, 66, 83, 85, 53, 106]
2	Query's result contains a set of tuples T	Formulate a query Q based on given example tuples T	[120, 119, 25, 69, 26, 110]
3	Remove outliers O in query's result	Given Q has outliers O in its results, refine Q to Q' to remove these outliers O	[104, 105, 129, 109]
4	Remove unexpected tuples U from query's result	Refine Q to Q' so that Q' removes unexpected tuples U	[118, 49, 48, 16]
5	Query's result satisfies aggregate constraint G	Refine query Q to Q' such that Q' satisfies constraint G	[78, 17, 2, 123, 125, 10, 11, 12, 57, 79, 14, 58]

Table 2.1: Summary of some QR techniques based on refinement constraints

of their databases' schema, since they provide alternative paths that are equivalent to that arbitrary query [120].

In other cases where users are unable to provide such tuples of interests, techniques such as AIDE [25] help in this regard by interactively steering users towards interesting data within a massive data space. To enable such interactive steering, AIDE carefully selects sample tuples for users to label as relevant or irrelevant. Then AIDE updates a classification model to further select sample tuples for users to label. Once users end this interactive steering process, AIDE formulates a query out of the classification model which captures users' interests.

2.2 Query Refinement Techniques

Informally, QR is *"the process of refining a query when the answer to the query does not meet the expectations of the user"* [17]. In particular, the *predicates* of the query are automatically *refined* so that the query's result reflects what the user expected, i.e., her *constraints*.

Refining predicates is tantamount to applying a set of modification operations on the predicates. These modification operations can be for example adding or dropping existing predicates, relaxing constants into ranges, narrowing ranges into constants, joining with auxiliary tables through foreign keys, etc.

Next, we review the literature of QR techniques based on the refinement constraints, which are certainly proposed to address a specific need or to solve a problem faced by users in DE tasks. We also

review the search and optimizations methods in these techniques. Table 2.1 above lists a summary of the discussed query refinement techniques broadly classified by the refinement constraints.

2.2.1 Query Refinement Techniques -Various Constraints

Too-few, Too-many, and Empty Answers Problems

In data exploration settings, it is not uncommon for users to experience the too-many, too-few or empty answers problems with their exploratory queries. These queries are often restrictive and narrow (too-few answers problem), liberal and under-specified (too-many answers problem), or they are unsuccessful in returning any answers at all (empty answers problem).

The empty answer problem is a special case of the too-few answers problem. These two problems aim to refine the original query into a new one, by applying modification operations on the predicates, so that the answer of the new query is likely to contain the tuples that interest the user. In [85, 83] they followed an interactive approach to solve this problem, i.e., users are asked for feedback on possible relaxation proposals for their queries. They proposed a probabilistic framework with exact and approximate algorithms which aim to refine a conjunctive query with atomic predicates by dropping some of these predicates (i.e., relaxation sequences) to achieve a non-empty-answer. To do that, all relaxation sequences are represented in a tree, rooted by the empty-answer query, and scored by the probability that a user accepts a proposed relaxation. Then, to find the best relaxation sequence, the FullTree [85] algorithm constructs this tree in full and recursively traverses it in a depth-first mode. Since this algorithm is computationally expensive and inappropriate for exploration settings, an optimized algorithm called FastOpt was proposed to minimize the construction cost of this tree by pruning branches of it using lower and upper bounds of the probability scores. These algorithms though were proposed to work with Boolean databases, i.e., attributes with 0s or 1s. They cannot be applied on databases that contain categorical and numerical attributes directly.

Using Machine Learning algorithms, [90, 91] developed LOQR: an online algorithm to relax a failing query (i.e., a query with an empty answer) with disjunctive predicates. The algorithm learns a decision rule for each attribute used in a predicate, then converts the learned rules into statements. The statements are then scored based on their similarity to the disjunctive predicates of the original query, and the most similar ones are used by LOQR to relax the predicates of the failing query.

The techniques in [106] also use Machine Leaning algorithms to address the too-many answers problem. They adopted the principles of the faceted search paradigm on structured databases, and proposed to take advantage of associated rich meta-data that comes in the form of tables, attributes, value ranges, etc. As an alternative method to ranked retrieval, faceted search was proposed to drill

down and zoom in on the tuples that interest the user, with least user effort. This search method is analogous to a dialogue with the user: a series of questions about the attributes are asked by the system, for which the user replies with a value. The result of this dialogue is a decision tree, where each node is an attribute and each edge leading out of it is assigned a value from that attribute's domain. The leaves of this tree are the tuples in the database, and a path from a leaf node up to the root is the dialogue the user had with the system, hence, the user effort is the average height of all paths in the tree.

Another work in [66] proposed a framework for relaxing a failing query that involves selection and join conditions. The framework defines the relaxation skyline as the set of all joined tuples that are not dominated by any other joined tuples. These skyline joined tuples are found by applying two steps: a join step (i.e., nested-loop join or hash-based join), and a skyline-computing step (i.e, block-nested-loops algorithm). To compute the skyline for a query with join conditions, it is assumed that there is a multi-dimensional indexing structure (e.g., R-Tree) for each attribute in the selection and join conditions. Then, an algorithm called MIDIR applies the two steps above (i.e., join and skyline-computing steps) on the objects in the R-Trees in a top-down fashion.

It is worth mentioning that the well-known Top-K [20, 47] and Skyline [22, 9, 44] operators were proposed to assist users in overcoming the too-many answers problem as well. With the help of a scoring function that scores all dimensions in a database, Top-K techniques rank the tuples of a query's result (i.e., the too-many answers) by aggregating the dimension's scores, then efficiently retrieve the top K scored ones to the user. On the other hand, the Skyline operator requires no scoring function to be defined for each dimension. Given a query that returns too many tuples, the Skyline operator returns the set of tuples that are not dominated by any other tuple in the query's result.

While the aforementioned techniques provide efficient and effective solutions in terms of their constraints, they partially address the defined constraints in the AQR problem. That is, these techniques provide no guarantee on achieving a target aggregate value, which is the goal of the AQR problem. Further, some of these techniques are limited to only one type of aggregates, i.e., `count` [90, 91], or can only be applied to Boolean databases [85, 83].

The 'Query by Examples' Problem

The problem of formulating a query using user-supplied tuples [120, 119, 5] or user-labelled tuples [25] has great applications in data exploration, and it is related, as well, to query refinement problems from the efficiency and effectiveness challenges sides. Although users do not provide failing queries to be refined, the process of formulating a query given tuples of interest requires traversing a huge

query space to find an optimal query. Further, determining the effectiveness of a solution seems challenging: either users are continuously asked for their feedback, or it is determined using the user-supplied information in the beginning of the query formulation process.

The applications of these formulated queries are diverse. One obvious application is to reinforce users' understanding of their databases schema by providing different paths (i.e., queries) that lead to tuples of interest. These queries might be simpler to understand, and uncover hidden relationships within complex databases. To find these queries, [119] models this problem as a data classification problem, and proposes a data-driven approach called TALOS. In TALOS, users provide a set of tuples (i.e., result of an arbitrarily query). Then, TALOS classifies the database tuples either positive or negative: if a tuple belongs to the result that the user provided, then it is classified positive, otherwise it is classified negative. TALOS then builds a decision tree based on these two classifications by iteratively splitting the tuples based on attributes until tuples in each leaf node are all positive or negative. The best split of an attribute is chosen as the attribute which splitting value maximizes a goodness criteria (e.g., entropy or Gini Index). Finally, the queries are formulated from the root-to-positive leaf nodes paths in the decision tree, and are ranked based on two variations of the popular F-measure metric. Since this approach incurs high computation costs, TALOS uses optimization techniques, such as optimizing joins with pre-computed join indices, to obtain a compact list of all joined tuples to efficiently construct the decision tree.

By considering queries with foreign key join conditions only, the work in [110] proposes a system with efficient techniques that produce valid queries for a given set of example tuples. A query is said to be valid if each example tuple appears in the result of this formulated query. The system has two main components. The first one is responsible for exhaustively formulating all candidate queries. Then, these queries are fed to the second component that is in charge of verifying these queries. This component applies different optimization techniques to speed up this process. For instance, it utilizes the dependency properties within these queries, i.e., if a query is pruned, then all of its sub-queries are also pruned, since they are more restrictive and cannot contain the tuples that their parent does not contain in the first place. A major limitation of their system is that it does not consider selection predicates for the formulated query, only join predicates based on foreign keys.

Another application for formulating a query using tuples is to steer users towards interesting regions in their data, then formulate queries that retrieve these regions [25]. For instance, AIDE [24] exploits sample selection and Machine Learning techniques in an interactive fashion to carefully select sample tuples for users to label, and continuously updates a model (i.e., a decision tree) that learns users interest. When the user decide to terminate the exploration process, that decision tree can be translated into a query. AIDE's main goal is to minimize the number of sample tuples that

users have to label relevant or irrelevant. To achieve that, AIDE follows three main phases. First, it extracts sample tuples for users to label by splitting each dimension into equal-width cells. Then, for each cell, the user is asked to label the tuple that is closer to the virtual centre of that cell. This process is repeated by dividing the cells that contain irrelevant tuples into finer-grained cells, since they might partially overlap with relevant areas. A classification model (i.e., decision tree) is repetitively fed with these tuples to learn the user interest and predicate relevant and irrelevant areas. The second phase is to polish the model by leveraging tuples that are labeled irrelevant by the user but are contained in a relevant area that was predicated by the model (i.e., false positivists), and tuples labeled relevant by the user but predicated to be irrelevant by the model (false negatives). The third and final phase is to refine the boundaries of these predicated relevant and irrelevant areas by selecting a much smaller sample size close to the boundaries for users to label. Since these three phases are dependent on extracting samples from the data, which can be an expensive overhead, AIDE applies two optimization techniques in that context. It uses a small sample of data, rather than the original data, generated using a simple random sampling approach that picks each tuple with the same probability. Also, it adaptively reduces the number of sample tuples to be labelled by users by remembering (at each iteration) the boundaries changes for each predicted area in the decision tree. Smaller changes between two consecutive iterations for a predicated area means the model has already achieved a reasonable approximation of that area, hence, new tuples most likely will not improve this area.

A slight modification of this problem is presented in [82, 84], in which users provide an exemplar query and the goal is to find other queries that have similar structures. This case is not uncommon in many practical scenarios where users are not aware of their interests' characteristics and structures. The proposed approach FastXQ in [82] represents the exemplar query as a connected sub-graph, and aims to efficiently find all isomorphic sub-graphs contained in the data graph by pruning nodes that are unqualified to take part in any candidate isomorphic sub-graph. To significantly reduce the search space, FastXQ represents the neighbourhood of every node (for both the exemplar query sub-graph and data graph) in a compact way, to avoid visiting all the nodes in the data graph.

The addressed refinement constraints in the above works are fundamentally different than the constraints defined in the AQR problem. Specifically, these constraints are defined on the individual tuples of the queries results, while in the AQR problem the constraints are defined over all tuples of the results. Hence, extending these works to address the constraints in AQR is not applicable.

Explain Outliers in Query Answer Problem

The related works in [104, 105, 129] consider the problem of finding explanations for outliers in their query answers. An example of these outliers can be an aggregate value that is abnormally different from the rest of the values in an answer produced by a group-by SQL query. The explanations that users are interested in are actually predicates that are added or removed from the original query (i.e., query refinement), which cause these outliers to disappear from the answer (i.e., refinement constraint). The work presented in [129] proposes a system called Scorpion to find these predicates by applying different partitioning and merging algorithms optimized by aggregate properties such as incrementally computing aggregate values, and pruning the search space using the monotonicity property of aggregate operators. In Scorpion, given a group-by query with P predicates which produces n aggregate values $\{g_1, g_2, ..., g_n\}$, the user is asked to populate two disjoint sets H and O from $g_i, 1 \leq i \leq n$ to flag the outlier values. Further, the user specifies an error vector V for O, where $v_i \in V$ can either be 1 if $g_i \in O$ is high, or -1 if $g_i \in O$ is too low. Hence, the aim is to find the *best* predicate $p^* \in P$ which explains the outlier values in O. To quantify that, a measure called Predicate Influence (PI) is proposed. The influence of a predicate p on an outlier aggregate value g is simply a ratio:

$$PI(p) = v\frac{\Delta g}{|p(g)|}$$

where Δg represents the change in the aggregate value after dropping predicate p, and $|p(g)|$ is the number of tuples in g that satisfy p. Recall that Scorpion aims to find the *best* predicate based on the PI definition. [129] formally defines the Influential Predicates problem, which aims to find the maximum influential predicate p^*:

$$p^* = argmax(inf(p)) \ , p \in P$$

where $inf(p)$ is defined as the average influences of p on the outlier values in O, minus the maximum influence of p among the holdout values in H. To find p^*, Scorpion performs two steps: partitioning and then merging. The first step exhaustively enumerates a list of all possible predicates P, and ranks them by their influence on O and H in a descending order. In the second step, Scorpion repeatedly merges adjacent predicates as long as the merging increases the influence. To address the efficiency challenges of these two steps, Scorpion utilizes several traditional properties of the aggregate operators, so they can be incrementally computed (i.e., additive operators) and they are monotonic, to optimize the partitioning and merging steps. For instance, Scorpion benefits from computing the aggregate value of a predicate incrementally from cached data in two ways: when

ranking the predicates in the first step, and when merging the predicates in the second step to evaluate the influence of the merged predicates.

Along the same lines of the Influential Predicates problem, [104] introduces a framework to explain outliers found in the result of multiple queries with foreign key joins. This type of complex queries enlarges the challenges in searching for explanations as the search space of predicates increases too. Similar to [129], an explanation e is a conjunction of atomic predicates on attributes, and outliers are aggregate values found in the result of a complex query. Although, different to [129], the framework proposes two methods to measure how well an explanation e is able to explain the outlier results. The first one is termed *degree of explanation by aggregation*:

$$e(v,Q) = \begin{cases} -Q(e), \text{ if } v = -1 \\ Q(e), \text{ if } v = 1 \end{cases}$$

If the user believes the aggregate in the query's result is too high (i.e., v=1), then the explanations (i.e., predicates) that produce the highest aggregate values are the top explanations for this high aggregate value, and vice versa. The second measure is related to their key novel contribution in this framework: the complex relationship between tuples that are joined by foreign key constraints, i.e., back-and-forth foreign key relationships. This causal path requirement is used to reduce the space of all possible explanations, since some of them become invalid under this causal path requirement. Anyhow, the second measure is based on intervention Δ and is termed *degree of explanation by intervention*:

$$e(v,Q) = \begin{cases} Q - \Delta_e, \text{ if } v = -1 \\ -(Q - \Delta_e), \text{ if } v = 1 \end{cases}$$

where Δ_e is the set of tuples that satisfy e. By restricting the queries to have atomic predicates with equality operators, [104] proposes an algorithm to find the explanations with the top degrees. The algorithm employs the data cube operator, which is supported in most commercial DBMS, to enumerate all predicates and to compute their aggregate values. This is done for each relation. The resultant cubes are then merged and the top explanations are returned. To illustrate this process, assume two relations $R_1(A,B,C)$ and $R_2(D,E,F)$ where each have two categorical attributes that the user is expecting to see the explanations from, and one numerical attribute used to compute an aggregate value g. This value can be, for instance, a ratio $g = agg(R_1.C)/agg(R_2.F)$. Let us assume the user thinks this ratio is too high. To find the top explanation for g being too high, the algorithm applies the cube operator on R_1 and R_2. Let us assume there are two

distinct values in $R_1.A = \{a_1, a_2\}$ and $R_1.B = \{b_1, b_2\}$, then the cube operator will return 2^3 rows: $\{(a_1,*,g),(a_1,b_1,g),(a_1,b_2,g),(a_2,b_1,g),(a_2,b_2,g),(*,b_1,g),(*,b_2,g),(*,*,g)\}$. The same for R_2. Each row corresponds to a possible candidate explanation (or a query), e.g., the row (a_1,b_1,g) is: `select` $agg(R_1.C)$ `from` R_1 `where` $A = a_1 \wedge B = b_1$;. The g in the row is precisely the *degree of explanation by aggregation*, i.e., the function $e(v,Q)$ defined above. However, before ranking these rows based on g, the algorithm needs to merge the rows produced by the cube operator from R_1 with R_2. Each row from R_1 is joined with all rows from R_2 (i.e, full outer join) to form a complete list of explanations that involve multiple relations, and the g value is recomputed. Finally, the top explanations are returned. To optimize the search for top explanations, the cube operator is run in advance for each relation to materialize the rows before joining them.

By regarding outlier values as aggregate constraints, the above techniques exhibit some limitations and shortcomings in addressing the AQR problem. For instance, both techniques [104, 129] do not directly specify a target value, rather they specify a direction, i.e., if a value is high or low. Moreover, [104] is applicable for queries with atomic equality predicates only. In case of range predicates (as defined in AQR), the cube operator may become extremely expensive to construct and to store, causing the algorithm's efficiency to decrease dramatically.

Refine for Expected and Unexpected Tuples

In [16], it is argued that query refinement techniques can be of great benefit in applications where users have limited access to the database, e.g., a web-based search engine for airlines tickets. In these applications, if a user's query is not returning the desired results, then it is difficult to manually alter the query's predicates and submit the query, as there are generally limits imposed by the database owner (e.g., number of queries per user) which makes manual refinement restricted. Motivated by that challenge, the work in [16] has proposed the *Why-not?* model which aims to identify the optimal set of predicates in a query Q's plan that are responsible for excluding a set of specific tuples T from Q's result. This optimal set of predicates is identified by representing Q's plan (i.e., predicates) in a directed acyclic graph, then finding out all *picky* predicates in that graph in a bottom-up or top-down approach. A predicate p is said to be *picky* if T is in p's input set but not in its output set. While there might be a large number of picky predicates, the optimal set of predicates, i.e., the *frontier picky predicates* are the ones at the top of the graph. That is, the last picky predicates to exclude T from the answer. These predicates are suggested to the user as answers for why T is not in Q's result.

The ConQueR approach in [118] was proposed to solve a similar problem: automatically and minimally refine Q to explain why-not questions for the missing tuples set T. Since there might be a

large number of refined queries that explain the why-not questions, the approach selects the refined query Q' that minimizes the imprecision and dissimilarity to Q. The imprecision of a refined query Q' to Q, with respect to the missing tuples T, is represented as the number of irrelevant tuples, and is given by:

$$\text{imprecision}(R(Q'), R(Q), T) = |R(Q') - R(Q) - T|$$

Where $R(Q)$ represents the set of tuples returned by query Q. Hence, the imprecision defined above measures how well Q' satisfies the refinement constraint. To guarantee minimal refinement, ConQueR requires that the dissimilarity of Q to Q' be minimal. The dissimilarity of Q to Q' is measured as the minimum edit distance to transform Q into Q'. Thus, the optimal refined queries are the set of all queries that dominate other queries in these two measures: imprecision and dissimilarity. That is, the skyline of the refined queries are the solutions for why not including T in the result of Q, which ConQueR finds in two phases. In the first phase, ConQueR modifies a predicate $P_i : A_i \leq c$ in Q to $P_i : A_i \leq c'$ such that c' is the maximum value between the corresponding values in attribute A_i in Q's result, and the corresponding value in attribute A_i in the missing tuple set T. By modifying all predicates in Q, this phase guarantees to return a refined query Q' including the missing tuples T. However, Q' will most likely contain too many irrelevant tuples as well. Hence, ConQueR improves the precision of Q' by adding selection predicates on attributes that are not present in Q's select clause. A selection predicate $P_j : A_j \leq c$ is added to Q' such that A_j does not appear in the selection clause of the original query Q, and c is the maximum value in A_j found in the set of tuples produced by Q when A_j is included in its projection clause, union the set of missing tuples T.

In [49], they proposed a formal framework for refining a query Q based on expected E and unexpected U tuples that users supply as a feedback. In this framework, users' feedback is further enhanced by finding implicit expected and unexpected tuples via finding the skyline of E and U in the remaining tuples $R - Q(R)$, where R is the set of all tuples in the database. Then, the predicates of Q are ordered based on their fitness score, which is computed using precision and recall. The fitness score for a given predicate in Q is computed by replacing its value with the corresponding attribute value in either E or U. A greedy approach is followed to select the predicates with the highest scores to apply the new values in Q.

Beyond the traditional SPJ queries, the works in [43, 50, 51] have addressed refining advanced queries such as Top-K and Skyline queries to include missing tuples. For instance, in [43], users provide a set of missing tuples T after reviewing the result of a Top-K query. Their approach in [43] is to minimally modify the result size K and/or the weighting vector W of the original query, such that the result of the refined query includes the missing tuples. Since W can be modified in

infinite number of ways, finding the optimal refined query is computationally challenging. Hence, their approach trades the quality of the refined query with the running time by considering only a sample of refined weighting vectors to find the best approximated refined query.

Along the same lines, the proposed technique in [50] aims to refine a reverse skyline query so that the answer of the refined query includes a missing point (i.e., a missing tuple). The refinement of a reverse skyline query in [50] is carried out on both the query point and the missing point. Specifically, to include a missing point in the result of a reverse skyline query, the missing point itself is moved within the data space so that it appears in the result of the reverse skyline query. Alternatively, the query point can be moved so that its result includes the missing point. The refinement approach in [50] ensures that the result of the initial reverse skyline query are retained in the refined query by allowing the movements of the query point to be within a safe region.

In other query settings such as the similar graph matching problem [51], users provide a set of missing graphs instead of tuples. The aim here is to find the optimal refined graph query that minimizes the distances between itself and the set of missing graphs, union the initial graph query answer. The constraints in the Similarity-aware, Aggregate-based Query Refinement problem addressed in this thesis differ from the constraint in the above problem in two ways. First, the objective in the problems addressed by this thesis is to minimize the combinations of both constraints: similarity and aggregate constraints. Users are able to change the weights of these constraints to reflect various applications requirements and needs. Second, the similarity constraint in the problems addressed by this thesis is measured as the difference between the predicates values of the initial query and the refined one, while in [51] the similarity of a refined graph query is measured as the maximum distance between this particular refined graph query and the graphs in its result, such that it is less than or equal to the maximum distance between the initial graph query and the graphs in its result.

In short, the QR problems addressed by the techniques above have different constraints to the Similarity-aware, Aggregate-based and Similarity-aware, Correlation-based Query Refinement problems addressed in this thesis. Extending those techniques to cater for the constraints in these two problems, and for the similarity constraint is simply not practical. For instance, in the Similarity-aware, Aggregate-based Query Refinement problem, users cannot provide expected, unexpected or example tuples. Rather, users provide aggregate constraints that a query must satisfy. A single aggregate constraint can be formed from a combinatorial number of tuples. Hence, it is not practical to ask a user to specify an aggregate constraint using tuples. Further, in this problem the constraints users specify for their queries are certain aggregate values, while in the above techniques the constraint is to include or exclude certain tuples. Similarly, the constraints and problem settings in the Similarity-aware, Correlation-based Query Refinement problem are quite different to the ones

discussed in the related work above.

Next, we discuss the shortcomings of the related work which addresses different varieties of the Similarity-aware, Aggregate-based (Section 2.2.2) and Similarity-aware, Correlation-based (Section 2.2.3) Query Refinement problems.

2.2.2 Aggregate-based Query Refinement Techniques

In AQR problems, users specify aggregate constraints (in a one-off interaction fashion), and the goal is to efficiently formulate or refine a query such that its result fulfills these user-specified constraints. The aggregate constraints can be cardinality constraints [79, 14], i.e., constraints over the size of the returned result, or aggregate values in the query result, e.g., [58]. The former has many applications, one of which is to obtain suitable query instances to evaluate new changes in a database engine, while the latter is useful in exploring data by searching for properties in the result, such as a specific aggregate value.

In databases testing, it is often required to evaluate the performance of a new component added to the database engine by running test queries before and after the component is added [13]. Generating these test queries though is challenging because their predicates have to be automatically examined and specified to serve a planned test scenario. For instance, it has been shown in [14] that generating a query given a single cardinality constraint over its result is NP-hard. By relaxing the requirement, i.e., accepting approximate solutions that are close from the cardinality constraints, they proposed a heuristics Hill Climbing approach. HC searches for the predicates values of a query that minimize the average relative error of the cardinality constraints:

$$\frac{1}{k}\sum_{i=1}^{k} max(\frac{c_i}{\hat{c}_i}, \frac{\hat{c}_i}{c_i})$$

where k, c_i and $\hat{c}_i$ are the number of cardinality constraints, the cardinality constraint of the sub-expression, and the current cardinality of the sub-expression, respectively.

In short, HC initializes the predicates by values that optimally minimize the relative error, as if they were independent. To move from this initial state, it refines the predicates' values one at a time by a step-size, and chooses the one that minimizes the relative error the most. These steps are repeated, and the step-size is halved if the relative error cannot be reduced by any further steps. HC terminates once it reaches a specific step-size.

The work in [79] addresses a general case of this problem where multiple cardinality constraints are defined. A practical solution called TQGen is proposed which can quickly generate queries that

approximately satisfy these multiple constraints. Specifically, TQGen returns a query that optimally minimizes the sum squared logarithmic relative error:

$$\sum_{i=1}^{k} (\log \frac{c_i}{\hat{c}_i})^2$$

Equipped with sampling based techniques and novel search procedures, TQGen is able to optimize the search and the number of calls made to the evaluation layer. The proposed solution TQGen works in two phases: firstly bounding the search space, then exploring this bounded space. The first phase outputs a query Q^U that overshoots all cardinality constraints (i.e., an upper-bound of the search space) by iteratively performing a binary search for each predicate over its domain. Then in the second phase, Q^U is explored efficiently to find optimal refined queries that minimize the relative error. This is done by partitioning all dimensions d in Q^U into k equi-width segments, producing a grid with exactly k^d cells. Thus, the potential test queries are defined as the intersection points of these cells boundaries. The partitioning of the grid is repeated recursively to obtain finer granularity grids, but only for the cells that are not pruned. A cell is pruned if its lower bound undershoots all cardinality constraints, or if the running sum of the lower bound exceeds the current best error. TQGen is frequently required to call the database evaluation layer to compute the relative error in these two phases. A single call to the database layer is considered expensive since it involves I/O operations, hence, repeatedly calling it will disadvantage TQGen as a practical solution. For that, TQGen uses two special sampling techniques that compute (in advance) super sets of the search space and store them into memory. Hence, relative errors are estimated from these memory-resident super sets, rather than calling the database layer.

A more complex instance of this problem is presented in [58]. This work proposes a new data exploration framework called Semantic Windows (SW) implemented on top of an existing DBMS. In SW, users search for windows of interests (i.e., queries) that are specified by content-based (e.g., aggregate values such as `avg(price) > $50`) and shape-based (e.g., a 2-dimensional rectangular shape) constraints, collectively called conditions. To find these queries efficiently, SW represents the search space as a grid: each dimension is divided into sub disjoint intervals of the same size, hence, the grid becomes a collection of disjoint cells. The goal of SW is to efficiently search this grid for adjacent cells that formulate a window (i.e., a query). A heuristic search algorithm (HOSA) was proposed to traverse this grid using a priority queue that orders the windows based on a utility score derived from the user-supplied conditions and I/O cost. This ordering resembles a cost-benefit analysis to rank the windows and ensure exploration of the ones that have higher utilities first. Specifically, given a

window w, its utility U_w is a combination of its cost and benefit:

$$U_w = B_w + (1 - \frac{C_w}{k})$$

The cost C_w is approximately the number of non-cached cells in w (k is the number of cached and non-cached cells in w), while the benefit B_w is how close w is to the defined conditions. HOSA employs sampling techniques to estimate B_w efficiently and filter out unpromising cells using shape-based conditions.

A major limitation of these techniques is that they are oblivious to the similarity of the refined or generated query to the user's initial query. Extending the techniques in [79, 14, 58] to cater for query similarity entails efficiency and effectiveness limitations. For example, since HC [14] chooses refinement steps based on evaluating the relative error locally, it is vulnerable to getting stuck at a local minima when query similarity is included in assessing the relative error of each step. While this might not be true for SW framework [58], it still suffers from high I/O and CPU costs from exhaustively evaluating all cells in the partitioned space when there are no shape-based conditions.

This thesis positions itself with [78, 17, 10, 2, 123, 125, 11, 12] since it shares with all of these works a similar assumption. This assumption is a common problem that users often face when performing DE tasks. Specifically, in some DE scenarios, users explore databases to retrieve desired results by formulating and submitting queries to the DBMS. A desired result, in this particular set of related works, is a target aggregate value that the query should return. Example 1.1 in the previous Chapter illustrates an instance of this exploration scenario.

With the assumption that users submit imprecise queries, AQR aims to automatically refine users' initial imprecise queries to maximize satisfaction of the aggregate constraints. While the AQR problem constraint seems somehow close from the first constraint in Table 2.1, it is fundamentally different. In Constraint 1, users do not explicitly provide a target value. Hence, the objectives are different in principal.

One of the earliest works on the AQR problem is the framework presented in [17]. This framework was proposed to partially automate the process of refining a query with a constraint on results, thus relieving the "undue burden" on users. It defines an extended query model as a doublet (Q,P), where Q is the input conjunctive query, and P is an acceptance test. The acceptance test P is a Boolean function which, given a query answer, returns either `true` or `false` to represent users' likelihood of accepting the answer (i.e., the constraint of refinement). An example of this function is:

$$P(Q) \equiv \texttt{count}(Q) \geq c.$$

The above acceptance test requires the answer to the query to be at least equal to c tuples, i.e., a minimum requirement of the answer's cardinality. Hence, the framework aims to minimally transform Q (a rejected query) to Q^* (an accepted query). To achieve that transformation, users have to define the acceptance tests P and associate them with modification operators (which users also have to define). Each operator is defined by a set of rewrite rules. These rules are precisely the refinement operations to be applied on the predicates that are mentioned in Section 1.3.1 above. For instance, a rewrite rule to generalize an answer to increase its cardinality is to remove a conjunctive predicate. A minimum refinement is guaranteed by selecting the minimum possible number of rules to obtain Q^*.

Similar to the generalization operator above, [57] addresses the problem of refining an initial query Q with d range predicates to a query Q^* that minimizes the absolute difference $|Q^*| - c$ where $|Q^*|$ is the number of answers returned by Q^* and c is the cardinality constraint, i.e., the `count(*)` aggregate operator. The problem also requires Q's answer to be smaller than c (i.e., similar to the too-few answers problem); the additional answers $\{Q^* - Q\}$ to be the closest to Q; $|Q^*| - c$ minimized; and the processing cost minimized. The SAUNA system guarantees the closeness of $\{Q^* - Q\}$ to Q by measuring the L_2-norm distance between the additional answers and the query Q and choosing the one with the minimum distance:

$$\sqrt{\sum_{i=1}^{k} d(p_i, Q)^2}$$

where $p_i \in \{Q^* - Q\}$ is a tuple in the additional answer set, and $1 \leq k \leq |Q^* - Q|$. Multi-dimensional histograms are used in SAUNA to address the processing cost involved in estimating the cardinality of candidate queries. However, each candidate query will have to be evaluated from the database at least once to compute its dissimilarity to the input query.

The Package Query problem defined in [10, 11, 12] is in some way an extension of the AQR problem presented in Section 1.3.1. Specifically, there are multiple target values a query has to satisfy, however, this problem does not consider the minimum refinement for an initial query, i.e., the user is interested in a query that satisfies the target values regardless of its similarity to the initial query Q. In a Package Query [11], the aggregate constraints are defined as global constraints, e.g., `sum(kcal) = 2000 and sum(saturatedfat) = 0`, which a query has to satisfy simultaneously. [12] proposes two algorithms to address this problem: one that returns exact solutions, and another that is more efficient and returns error-bounded approximate solutions. Both translate the global constraints into an ILP problem, and use one of the common ILP solvers (e.g., IBM CPLEX) directly to find solutions for the ILP problem. The approximate version SketchRefine is based on the divide-and-conquer approach to divide the search space (offline) into partitions represented by approximate tuples, and then apply ILP solvers on each one of these partitions to obtain a sketch of the solution. Then the

approximate tuples are replaced with the real data and the ILP solvers are applied again on each partition to obtain the final refined output, i.e., tuples that maximally satisfy the global constraints. Note that these algorithms search for optimal tuples, rather than queries, which is a major difference to the goal of the AQR problem, where a query is in a conjunctive form.

The related works in [78, 123] specify the cardinality of the query's result to be the refinement constraint (i.e., `count(*)`) while [125] generalizes the constraint to be any standard aggregate value, i.e., `sum, avg, min, max`. The problem defined in [123, 125] addresses a constraint similar to the AQR problem. The constraint is for the query result to satisfy an aggregate constraint, e.g., $\texttt{agg}(a_i)$, where `agg()` can be the result of applying any aggregate operator on any numerical attribute a_i. The previous techniques discussed in Section 2.2.1 support no such constraint.

The ACQUIRE approach in [125] accepts from the user an aggregate query Q that is to be refined, an aggregate constraint c, and a threshold φ, and aims to find all refined queries $q \in Q^*$ that are within the user-specified threshold, i.e., $|c - q| \leq \varphi$ for all $q \in Q^*$. To find Q^*, ACQUIRE partitions the search space (starting from Q) using a constant step size, and then iteratively enumerates and navigates the possible refined queries based on their closeness to Q. A refined candidate query is added to the solution if it is within the user threshold φ. To check this condition, ACQUIRE is actually required to compute the aggregate. It does so by computing the aggregate incrementally from previous queries, using the additive property of the aggregates. This is a core principle in the ACQUIRE approach which is true for aggregates such as `count, sum, max, avg`. ACQUIRE extensively uses this additive property of aggregates while navigating the search space. However, ACQUIRE ignores the I/O costs endured while navigating the search space. Further, it uses a fixed step size during its iterative enumeration of candidate queries, which could reduce the algorithm's efficiency if the solution is located far away from Q.

Given an inconspicuous query Q which returns too many/few answers, the SnS framework in [78] interacts with the user to relax or contract this inconspicuous query by transforming the predicates in one direction only, i.e., either expanding the predicates, or contracting them. This is a restrictive problem of AQR where the refinement constraint is the cardinality of the query's result, i.e., `count(*)`. The SnS framework refines Q in two steps. First, it bounds the search space by performing a binary search iteratively on each predicate in the original query Q while fixing the remaining predicates. The goal of this step is to secure a superset query Q^U that is guaranteed to have the solution. Then, in the second step, this superset query (which corresponds to a d-dimensional rectangle) is navigated interactively with the user, one predicate at a time in a rounds fashion. In each round, the user arbitrarily selects a predicate, and manually refines it within its new artificial bounds (i.e., within the d-dimensional rectangle found in the first step). The framework then responds by

performing a binary search on all predicates that are yet to be refined by the user. This results in shrinking the bounds of the unrefined predicates, hence, making it easier for users to select a value within a tight range rather than the original range of the predicates. During these steps, SnS estimates the cardinality using a random sample with bounded approximate error that is constructed once for each query. This sample enables SnS to provide fast and accurate estimation during the interactive navigation. Although SnS framework manages to incorporate users' preferences in refinement, it does not provide a fully automatic solution for refinement since the second step requires users' feedback for all predicates.

To summarize, although some of the aforementioned techniques provide efficient and effective techniques for special cases of the AQR problem, they exhibit certain limitations. For instance, the techniques in [79, 14] do not consider the similarity of the refined query to the input query, and are limited to one special aggregate constraint: cardinality. Similarly, the SW framework and the Package Query in [58, 10] do not account for the refined query's similarity. Moreover, the SW framework depends on only shape-based conditions to prune the possibly exponential search space. For the techniques that considers the refined query's similarity in refinement, the framework in [17] requires an experienced user to define rewrite rules for each refinement operator, which can be an obstacle especially for users with no background in the data. The SnS framework [78] requires no such rewrite rules, however, it fails to automatically infer the refined query's similarity to the input query. That is, it obtains users' feedback on each predicate iteratively. Further, it is limited to one special case of aggregates. Being a fully automatic solution, ACQUIRE [125] is able to return refined queries with any aggregate constraints efficiently by utilizing the additive property of aggregates. In particular, it partitions the search space based on a constant step size then exhaustively examines all candidate queries in this partitioned space based on their closeness to the input query Q. However, there are two main limitations: 1) although an aggregate of a candidate query is computed incrementally from previous queries, the I/O cost was not addressed, and 2) the constant step size depends on parameters that users have to specify, and could reduce the algorithm's efficiency if the solution is located far away from Q.

2.2.3 Correlation-based Query Refinement Techniques

As mentioned briefly in Section 1.3.2, the CQR resembles the Query by Output and Query Reverse Engineering problems presented in [119, 120, 115]. The goal in solving these problems is: given an output of a query, find the original query that returns the exact same output. Similarly, in the CQR problem the user provides an output (i.e., target pairwise correlation values) and the aim is to refine

the user's input query until this target is achieved.

However, the pairwise correlation constraints and the time series data in CQR introduce unique challenges which cannot be addressed by the techniques proposed there. For instance, [119] requires that the query's result must not contain an arithmetic expression such as correlation, whereas [115] optimization techniques are based on series of rules using aggregates' properties that are not applicable to correlation.

A more related body of work is presented in [70, 107, 87, 39] where the focus is to compute the correlation efficiently. Although all of them do not consider the similarity notion, they are tightly related to the Similarity-aware, Correlation-based Query Refinement problem, because computing correlation efficiently is at the core of this problem.

The BRAID algorithm in [107] utilizes the fact that correlation can be computed incrementally to quickly detect any pairs that are highly correlated, with a slightly small storage overhead. To illustrate how correlation is computed incrementally, let us look at the Pearson's correlation coefficient function for a pair of time series T_i, T_j of equal length ℓ:

$$\rho(T_i, T_j) = \frac{\ell \sum_{k=1}^{\ell} v_k^i v_k^j - \sum_{k=1}^{\ell} v_k^i \sum_{k=1}^{\ell} v_k^j}{\sqrt{\ell \sum_{k=1}^{\ell} (v_k^i)^2 - (\sum_{k=1}^{\ell} v_k^i)^2} \sqrt{\ell \sum_{k=1}^{\ell} (v_k^j)^2 - (\sum_{k=1}^{\ell} v_k^j)^2}}$$

where v_k^i is the kth value in T_i.

The five summation components (referred to as sufficient statistics) $\sum v_k^i v_k^j$, $\sum v_k^i$, $\sum v_k^j$, $\sum (v_k^i)^2$ and $\sum (v_k^j)^2$ are computed from scratch once, and then cached in memory. When a new data point is added to the time series T_i and T_j, the cached sufficient statistics are updated by adding this new data point to the five summation components. Then, the correlation $\rho(T_i, T_j)$ is computed in a constant time.

The techniques in [70, 71] build on the observation of [107] where correlation can be computed incrementally. These techniques however are a special case of Similarity-aware, Correlation-based Query Refinement problem since there is only one correlation constraint for all time series pairs, and they compute the correlation for pairs that are coupled with a given time series.

The work in [70] focuses on the problem of finding the longest time interval in which a pair of time series are highly correlated. This is clearly a limited version of the Similarity-aware, Correlation-based Query Refinement problem discussed above in Section 1.3.2. Specifically, the problem in [70] is a Bi-variant analysis problem, whereas the Similarity-aware, Correlation-based Query Refinement problem is a Multi-variant problem. Moreover, there is only a single threshold (i.e., a single correlation constraint) in [70], while in CQR there are as many correlation constraints

as the number of time series pairs. The SKIP algorithm proposed in [70] exhaustively searches all time intervals in a top-down strategy to find the longest, highly correlated interval of two time series. Similar to [88], SKIP precomputes the sufficient statistics of all time series for all possible lengths of time intervals (provided that the starting position of all intervals is the first data point in each series), and cached them in memory. Then, to compute the pairwise correlation for any length in a constant time, SKIP retrieves the corresponding cached statistics and performs the required addition and subtraction operations. Suffering from a large storage overhead for caching the sufficient statistics for all time series for all possible lengths, a technique is proposed which trades off the storage overhead of these statistics and the time it takes to compute the Pearson's correlation.

By following a different execution strategy than SKIP, [71] proposes an algorithm called ZES which is superior to SKIP. Under the ZES algorithm, there is no need to precompute the sufficient statistics for all possible lengths of time intervals. The ZES algorithm's execution strategy enables it to compute the correlation in a constant time by updating the cached sufficient statistics incrementally, similar to BRAID algorithm [107].

The AEGIS framework in [39] was proposed for fast computation of correlation in a distributed environment. The framework's goal is to report all pairs of all time series that are correlated above a certain correlation value, i.e., it assumes only one correlation constraint for all pairs. This framework heavily depends on this single constraint. Further, it assumes the length of the time sub-interval is known prior, e.g., given by a user. Based on these two assumptions, the framework partitions the time series such that the correlated pairs are contained in a single partition, or its neighboring partitions, to minimize communication costs. Extending this framework to address the general case in the Similarity-aware, Correlation-based Query Refinement problem where none of the two assumptions are enforced is not applicable.

A similar work is presented in [87], however, it also enforces the same assumptions as in [39]. [87] proposes to transform the time series into the frequency domain using Discrete Fourier Transformation (DFT) to control the CPU and I/O costs in computing pairwise correlation for all series of a given length. Using the transformed series, their algorithm partitions the series into batches, such that each batch contains highly correlated series. Batches with highly uncorrelated series are pruned and are not loaded into memory, i.e., reducing I/O costs. To reduce CPU costs (i.e., computational costs), [87] leverages the DFT representation of the original time series and its linear transformation to correlation (z-normalized Euclidean distance) to avoid computing correlation for all pairs. Specifically, the algorithm initially checks the distance between the DFT representation of a pair of series. If that distance is lower than a specific threshold (i.e., correlation constraint), then this pair is pruned and the algorithm skips computing its correlation value.

A more recent work is presented in [100] which aims to report highly correlated pairs while minimizing the response time. To achieve low latency, the proposed algorithm in [100] follows the same incremental approach as in [107] to incrementally compute the correlation of all pairs. A priority-based search algorithm is also proposed in [100] which orders the computations of pairs such that the ones that are highly correlated are computed first.

In summary, although the above techniques provide efficient algorithms for computing correlation, the assumptions enforced (e.g., length of time sub-interval is known, one correlation constraint for all pairs, Bi-variant analysis) render them short in addressing the goal of the Similarity-aware, Correlation-based Query Refinement problem. In particular, the Similarity-aware, Correlation-based Query Refinement problem discussed in Section 1.3.2 presumes that the user is oblivious to the starting location and length of the time sub-interval. Moreover, it presumes that there is a correlation constraint for each pair of time series. Hence, the goal is to refine the user's query (i.e., the time sub-interval) so that the query's result satisfies these pairwise correlation constraints, while maximizing the similarity between the refined query and the original one.

CHAPTER 3

Similarity-aware Aggregate-based Query Refinement

3.1 Overview

This chapter presents our contributions towards the Similarity-aware, Aggregate-based Query Refinement problem: inclusion of similarity in query refinement. It is organized as follows: Section 3.2 provides notations and definitions for the Similarity-aware, Aggregate-based Query Refinement problem which are used throughout the sections of this chapter. These notations and definitions include the cost model assumed in refinement, the definition of query similarity and its measures, a declarative model for query refinement and the formal problem definition.

Section 3.3 introduces a special case of the Similarity-aware, Aggregate-based Query Refinement problem where the aggregate constraint is the cardinality of the result. Accordingly, innovative schemes called SAQR are presented in Section 3.3 which aims to efficiently refine queries based on cardinality constraints on the result.

In Section 3.4, the general case of the Similarity-aware, Aggregate-based Query Refinement problem is presented where users' constraints can be any SQL standard aggregate operators `sum`, `avg`, `min`, `max`. For that general case, efficient approximation schemes are proposed and compared to related algorithms.

Lastly, Section 3.5 presents a web-based application *ORange* which employs SAQR schemes for refining selected areas based on cardinality constraints.

Symbol	Description
I	Input query
B	Database B
$P_i : a_i \leq x_i^I$	Predicate P_i on numerical attribute a_i in input query I
w_i	Weight of predicate P_i
R	Refined query
G_R	Aggregated value of R
G	Aggregate constraint
Δ_R^G	Aggregate deviation of R
Δ_R^S	Similarity deviation of R
α	Similarity weight
Δ_R	Total deviation of R
δ	Granularity of the search space

Table 3.1: Summary of Symbols

3.2 Notations and Definitions

In this section, we firstly introduce notations and definitions for the Similarity-aware, Aggregate-based Query Refinement problem, then we present the formal definition. All used symbols are listed in Table 3.1 with their descriptions.

Assume the presence of a database B with one or more relations that are linked by foreign keys. The input to the refinement process is an initial select-project-join (SPJ) query I, which is to be transformed into a refined query R.

A query I is a conjunctive, first-order query defined in terms of d predicates $P_1, P_2, ..., P_d$ over d numerical and categorical attributes in B. Similar to [78, 108, 55, 109, 12], we assume in our model that all relevant relations in B are joined beforehand by foreign keys, i.e., B is a d-dimensional flat relation that is materialized in advance by joining all relations through foreign keys.

Numerical Attributes: A predicate P_i on a numerical attribute a_i is in the form $l_i \leq a_i \leq u_i$, where a_i is the ith attribute in B, and l_i and u_i are the lower and upper limits of query I along attribute a_i, respectively. This results in a range query represented as a d-dimensional box (also known as hyper-rectangle or orthotope).

The domain of each attribute a_i is limited by a lower bound L_i and an upper bound U_i. Hence, a predicate P_i can be further expressed as: $L_i \leq l_i \leq a_i \leq u_i \leq U_i$. We note that an attribute a_i that is not included in I, is equivalent to $L_i \leq a_i \leq U_i$.

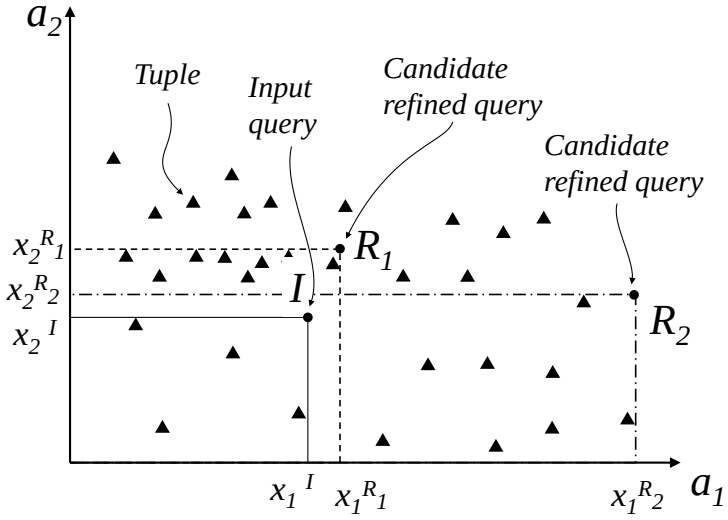

Figure 3.1: Example - refining an input query I in a two-dimensional space

A refined query R for an initial query I is achieved by modifying the lower and upper limits for some of the predicates in I. That is, for a predicate P_i in query I, a refined predicate P_i' in R takes the form $l_i' \leq a_i \leq u_i'$.

Similar to [79], we assume that a double-sided predicate is equivalent to two separate single-sided predicates. Hence, a predicate $l_i \leq a_i \leq u_i$ is equivalent to the following two predicates: $a_i \leq u_i$, and $-a_i \leq -l_i$. Accordingly, refining a single-sided predicate $a_i \leq x_i$ on a numerical attribute takes place by means of one of two operations as follows:

1. **Predicate expansion:** in which $a_i \leq x_i'$, where $x_i' > x_i$, or
2. **Predicate contraction:** in which $a_i \leq x_i'$, where $x_i' < x_i$.

Thus, under both of the two operations, the value $|x_i - x_i'|$ is the amount of refinement applied to predicate P_i. For example, an input query I:

I: `SELECT * FROM B WHERE` $a_1 \leq x_1^I$ `AND` $a_2 \leq x_2^I$;

can be refined be expanding its predicates across the two attributes a_1 and a_2, as shown in Figure 3.1:

R_1: `SELECT * FROM B WHERE` $a_1 \leq x_1^{R_1}$ `AND` $a_2 \leq x_2^{R_1}$;

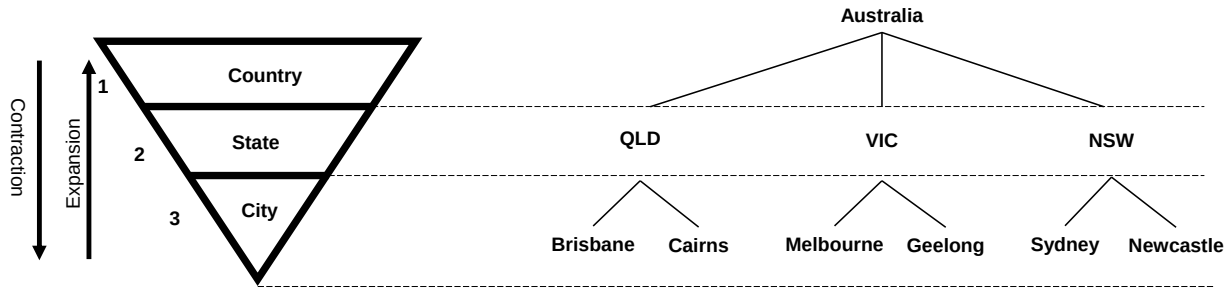

Figure 3.2: A categorical attribute `location` represented as a three-levels hierarchy to enable refinement. Each value in `location` is mapped to a level, i.e., `city, state, country`

R_2: `SELECT * FROM B WHERE` $a_1 \leq x_1^{R_2}$ `AND` $a_2 \leq x_2^{R_2}$;

Categorical Attributes: In the presence of categorical attributes, a multi-level hierarchy is typically used to rank the different categorical values. Hence, refining a categorical predicate of this form $P_i : a_i = x$ is simply mapped to moving up or down within the hierarchy [78, 125]. For example, consider a categorical attribute called `location` that indicates the placement of a product in a warehouse database in Australia. A three-levels hierarchy is created to map the values of the `location` attribute to specific ranks: `city, state, country`, which facilitate the refinement of that predicate. As shown in Figure 3.2, moving up in the hierarchy is equivalent to expanding the predicate, while moving down is obviously the process of contracting that predicate.

Join Predicates: Join predicates are excluded from refinement since these predicates are specified on identifier attributes (i.e., primary key, foreign-key attributes). A join predicate P joins two relations R_1 and R_2 on a common attribute a_i, i.e., $P : R_1.a = R_2.b$. Clearly, join predicates of this form do not follow the numerical predicate form defined earlier. Hence, the two refinement operations are not applicable on these predicates.

Clearly, the number of possible refined queries is exponential in the number of dimensions and forms a combinatorial search space. For instance, consider a query I over a d-dimensional database, in which each attribute a_i is discrete and the number of distinct values in each dimension a_i is n. For query I, the set of possible refined queries form a query space $\mathbb{R}$, where the size of that space is $|\mathbb{R}| = n^d$.

Given an objective for query refinement (e.g., satisfying a certain aggregate constraint), exploring the large search space $\mathbb{R}$ to find the optimal parameter settings (i.e., optimal R) becomes a non-trivial and challenging task. For instance, it has been shown that finding an optimal query R with cardinality

constraints (a special case of aggregate constraints) is an NP-hard problem [14].

To circumvent the high-complexity of query refinement, several search heuristics have been proposed (e.g., [78, 79, 14, 57, 123]) which aim to minimize the deviation between the aggregate constraint and the achieved one. It has been shown that these search heuristics often provide efficient and effective solutions to this Aggregate-based Query Refinement problem. In particular, the relationship between the possible refined queries and their corresponding constraints exhibit a *monotonic* pattern [14], which makes it practical to apply local search methods. As expected, the main idea underlying these heuristics is to limit the search space to a small set of possible candidate queries $\mathbb{R}_c$, such that $\mathbb{R}_c \subseteq \mathbb{R}$, and $|\mathbb{R}_c| \ll |\mathbb{R}|$.

3.2.1 Cost Model

For each candidate query $R \in \mathbb{R}$, a probe of the database is required to estimate the aggregate of R. Current techniques use alternative methods to perform such probe such as sampling [78, 79], and histograms [14, 57].

Irrespective of the employed estimation method, a call has to be issued to the database evaluation layer, where the aggregate value of R is estimated by running it on a small set of data (i.e., a sample or histogram table). This makes the probing operation inherently expensive and is a strong motivation for reducing the total number of probes required to achieve the optimal R. Accordingly, the incurred *cost* of the refinement process is measured in terms of the number of probes to the evaluation layer, and is defined as:

$$C_R = |\mathbb{R}_c| \tag{3.2.1}$$

We also follow the same approach as in [14, 79], in which we consider the probing operation for aggregate estimation as a *blackbox* and our goal is to minimize the number of such probes. Differently from those approaches, however, our goal is to achieve a Similarity-aware, Aggregate-based Query Refinement, as we describe next.

The techniques mentioned above are oblivious to the (dis)similarity between the input query and its corresponding refined version. That is, to meet the aggregate constraint, the generated refined query might often be very far (i.e., dissimilar) from the input query. While the (dis)similarity between two queries can be quantified using several alternative measures (e.g., [118, 98, 57, 90, 123]), it should be clear that, irrespective of the adopted similarity measure, a refined query that is very different from the input one will have a very limited benefit to the end user and is often rendered useless [118].

Therefore, we propose the Similarity-aware, Aggregate-based Query Refinement problem, in which the user satisfaction is measured in terms of both:

1. Meeting some specified aggregate constraint on the refined query, and
2. Maximizing the similarity between the input query and its corresponding refined one.

Achieving such goal is rather a challenging task as it would require an exhaustive examination of the large space of possible query refinements. In particular, the monotonic property exhibited under the objective of Aggregate-based Query Refinement no longer holds for our dual-criteria objective, in which aggregate constraints are combined with similarity. Hence, current Aggregate-based Query Refinement techniques that are based on local search heuristics (e.g., the Hill Climbing algorithm in [14]) have higher chances of meeting a local minima and falling short in finding a refined query that strikes a fine balance between minimizing the deviation in aggregate constraints and maximizing the similarity.

Before presenting the Similarity-aware, Aggregate-based Query Refinement problem, we discuss in detail how to measure the similarity of a refined query, and the tradeoff between efficiency and effectiveness that is manifested in the query similarity measures.

3.2.2 Query Similarity Measures

The intuition of including query similarity in refinement is to provide a refined query R that is as similar as possible to the input query I. A refined query R that is similar to the input query I will most likely retain as much of the user's intention in the input query I and ultimately increases the user's satisfaction with the results.

Box Query Similarity Method	**Examples**
Predicate-oriented	[118, 60, 61, 17]
Data-oriented	[98, 57, 117]
Value-oriented	[90, 123, 125]

Table 3.2: Examples from the literature for box query similarity measures

Measuring the (dis)similarity between two *point* queries is very well-studied in the literature, where typically a variant of the L_p-norm metric is used for that purpose (e.g., $p = 1$, or $p = 2$ for measuring the Manhattan, or Euclidean distances, respectively). Meanwhile, there is a lack of an established standard for measuring the distance between two box queries (i.e., I and R), which are the building blocks for the query refinement process.

We broadly classify existing methods for measuring the distance between two box queries as: predicate-oriented, data-oriented, and value-oriented methods (Table 3.2).

In the predicate-oriented measures (e.g., [118, 60, 61, 17]), the distance between I and R is mapped to that of measuring the *edit distance* needed to transform I into R, where the set of allowed transformation are: add, delete, or modify a predicate. Because of its simplicity, a predicate-oriented measure is very coarse for the purpose of query refinement as it falls short in distinguishing between the different possible modifications that can be applied to each predicate. That is, refining predicate $a_i \leq x_i^I$ into $a_i \leq x_i^R$ counts as one modification operation regardless of the value x_i^R and the amount of refinement, i.e., $|x_i^I - x_i^R|$.

In the data-oriented measures (e.g., [98, 57, 117]), the distance between I and R is based on the data points (i.e., tuples) that are included in the result of each query. For instance, to measure the distance between I and an expanded R, [57] computes the distance between I and all the points in $R - I$ (i.e., the extra points added due to expansion). Clearly, data-oriented methods incur a large overhead, which potentially renders a query refinement process infeasible.

Finally, in the value-oriented measures (e.g., [90, 123, 125]), the distance between I and R is based on the amount of refinement experienced by each predicate. Formally,

$$D(R,I) = \frac{1}{d}\sum_{i=1}^{d}\frac{|x_i^R - x_i^I|}{U_i - L_i} \tag{3.2.2}$$

Compared to the predicate-based methods, Eq. 3.2.2 considers the amount of applied refinement (i.e., $|x_i^R - x_i^I|$) and provides a reasonable approximation of the data-oriented measures at a negligible cost. Those reasons render Eq. 3.2.2 to be a suitable choice for the Similarity-aware, Aggregate-based Query Refinement problem considered in this thesis.

Often, however, users have partial preferences over what predicates to refine and by how much. That is, the individual weights of query I's predicates $P_1, P_2, ..., P_d$ on the objective of refinement, for a particular user, are uneven [132]. Accordingly, we introduce a new parameter to control the degree of importance for each predicate, from a user's point of view. We define w_i as a user-supplied weight for predicate P_i, such that its value is within the range (0-1), where a value of one means predicate P_i has the highest possible level of importance, while on the other hand a value of zero means it has no importance at all. We incorporate the weights and rewrite Eq. 3.2.2 to be:

$$D(R,I) = \frac{1}{d}\sum_{i=1}^{d}\frac{|x_i^R - x_i^I| \cdot w_i}{U_i - L_i} \tag{3.2.3}$$

Where $\sum w_i = 1$ for $i = 1, ..., d$. We note that for the sake of simplicity, we assume (from now on) all predicates are equally important, i.e., they have equal weights: $w_i = \frac{1}{d}$ for $i = 1, ..., d$.

Accordingly, we use the function expressed in Eq. 3.2.3 as the default measure of distance

for range predicates in the Similarity-aware, Aggregate-based Query Refinement problem, which is formally defined below.

3.2.3 Problem Definition

The current query refinement techniques (e.g., [79, 14, 123, 78]) often provide efficient and effective solutions to the Aggregate-based Query Refinement problem. Specifically, these techniques aim to quickly navigate a large search space of possible refined queries, and return one refined query such that its aggregate is very close to a specified aggregate constraint. That is, minimize the deviation between the aggregate constraint and the achieved one. Formally, given an aggregate constraint G, the aim of these techniques is to find a refined query R such that Δ_R^G is minimized:

$$\Delta_R^G = \frac{1}{z}|G - G_R| \tag{3.2.4}$$

where z is a normalization factor, and G_R is the aggregate value of R.

Clearly, while the user might be satisfied that the refined query's aggregate value G_R is very close to the constraint G, they would also expect the refined query R to be very close (i.e., similar) to their input query I. A refined query that is very different from the input one will have very limited benefits to the end user and is often rendered useless.

Motivated by that, we propose the Similarity-aware, Cardinality-based Query Refinement problem, in which the user satisfaction is measured in terms of both: 1) meeting some user specified aggregate constraint on R, and 2) maximizing the similarity between R and I. Formally:

Definition 3.1. ***Similarity-aware, Aggregate-based Query Refinement problem:*** *Given a database B, an input conjunctive query I, a distance function $D()$, and an aggregate constraint G over the result $I(B)$, the goal in the Similarity-aware, Aggregate-based Query Refinement problem is to find R that satisfies the aggregate constraint G while minimizing $D(R,I)$.* ■

Ideally, the distance between R and I (i.e., $D(R,I)$) should be equal to zero (i.e., maximum similarity such that $R \equiv I$). In reality, however, achieving that extreme case of exact similarity is unrealistic, unless query I already satisfies the aggregate constraint G, i.e., $G = G_I$. That is, I already meets its aggregate constraint G and no further refinement is required. Hence, in this work, we adopt a hybrid metric, which captures and quantifies the success of meeting the user's expectations for both similarity and aggregate constraints. In particular, we capture the user's (dis)satisfaction in terms of the overall deviation (in both, aggregate and similarity constraints) from the user's expectations, which is formally defined as:

$$\Delta_R = \alpha\Delta_R^S + (1-\alpha)\Delta_R^G \tag{3.2.5}$$

where Δ_R^S is the deviation in similarity, which is captured by means of a distance function $D(R,I)$ as described earlier in Section 3.2.2, and Δ_R^G is the aggregate deviation defined above in Eq. 3.2.4.

The parameter α simply specifies the weight assigned to the deviation in similarity, and in turn, $(1-\alpha)$ is the weight assigned to the deviation in cardinality. Parameter α can be user-defined so as to reflect the user's preference between satisfying the aggregate and similarity constraints.

On the one hand, setting $\alpha = 0$ is equivalent to the AQR problem. On the other hand, setting $\alpha = 1$ is equivalent to the extreme case described above, in which $R \equiv I$. In the general case, in which $0 < \alpha < 1$, both the aggregate and similarity constraints are considered according to their respective weights and the overall deviation is captured by Δ_R. Hence, a small value of Δ_R indicates a small deviation in meeting the constraints, and more satisfaction by the refined query R.

Interestingly, the similarity and aggregate constraints are typically at odds with each other. That is, maximizing similarity (i.e., Δ_R^S) while minimizing aggregate deviation (i.e., Δ_R^G) are two objectives that are typically in conflict with each other, and α specifies by how much those two constraints contribute to the overall deviation Δ_R. For instance, assume that the input query I in Figure 3.1 does not satisfy an aggregate constraint G. In order to satisfy G, query I has to be refined by expanding or contracting its predicates. Hence, any refined query R that minimizes Δ_R^G (i.e., closes on the constraint G), will have to increase Δ_R^S (moves far from I by expanding or contracting its predicates).

Lemma 1. *Minimizing Δ^S always conflicts with minimizing Δ^G, provided that $0 < \alpha < 1$ and the input query does not satisfy the aggregate constraint.*

Proof. It is trivial to proof the above lemma using the example in Figure 3.1. Assume that I does not satisfy an aggregate constraint G, e.g., the cardinality of I is less than a cardinality constraint G. Hence, to satisfy G (i.e., minimize Δ^G), query I has to be refined by expanding its predicates. This will always result in increasing Δ^S. ■

As an alternative method for deciding the appropriate α in Eq. 3.2.5, user feedback can be used to infer the suitable α [77, 111]. For instance, users are asked to label the result of a small sample of refined queries as *acceptable* or *unacceptable*. These refined queries are the result of refining a query with an aggregate constraint using different values of α, e.g., 0.1, 0.4, 0.6, 0.9. Then, the preferred α can be inferred from these labeled queries. The larger the sample is, the better value of α is, however, the more queries users have to label.

Alternatively, α can be system-defined and is set automatically to meet certain business goals or

objectives that are defined by an application. For instance, assume an application that shows exactly K number of flights for any user-selected departure and destination within a user-selected time slot. If no flights are found within the user-selected options (i.e., the input query returns empty result), then it is more popular and common to show alternative similar flights than showing an empty screen, which meets the application's goal. Hence, the input query can be refined with a suitable value of α so that the refined query returns almost K number of flights that are very similar to the input query and can fill the application's screen by suggested flights for the user.

Before presenting our proposed techniques for solving the problem defined in Def. 3.1, we present a declarative query model which encapsulates all the essential parameters that are used by our proposed techniques.

3.2.4 Declarative Query Model

To support query refinement for a large spectrum of database users, we propose a model with a friendly command interface which essentially encapsulates all usage scenarios and sits as a medium between users and the proposed schemes. This model is an extension of the traditional SQL language to capture users' constraints for refinement. Specifically, the user-supplied constraints are enclosed within a new clause, such that the refined query must satisfy all of them. Here is the proposed, extended query structure:

```
SELECT * FROM <relations> WHERE <predicates> WITH_CONSTRAINTS
SIMILARITY α = <X> AND
DISTANCE_FUNCTION = <Y> AND
<AGG_OP>(<attribute>) = G;
```

The new keyword `WITH_CONSTRAINTS` indicates that there are user defined constraints over the result of the query. The first user-defined constraint is the weight of similarity (i.e., α), where `<X>` is a value between (0-1).

The second parameter in the model is the choice of a distance function `<Y>`. The distance function (i.e., $D()$) is used as a measure of the similarity deviation between an input query I and a refined query R, where a high distance corresponds to a high similarity deviation, and vice versa. The similarity measures have been discussed above in Section 3.2.2.

Lastly, there is the aggregate constraint `<AGG_OP>(<attribute>) = G`. From the five standard SQL aggregate operators (i.e., `count, sum, avg, min, max`), users can specify an aggregate operator for the aggregate constraint G over any attribute in the database. Also, users are allowed to specify multiple aggregate constraints at once:

$$\texttt{<AGG\_OP>}_1(< attribute >) = G_1$$
$$\texttt{<AGG\_OP>}_2(< attribute >) = G_2$$
$$..$$
$$..$$
$$\texttt{<AGG\_OP>}_n(< attribute >) = G_n$$

With the presence of multiple aggregate constraints, the deviation function in Eq. 3.2.4 must be

extended to account for all n aggregate constraints:

$$\Delta_R^G = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{z}|G_i - G_{iR}| \tag{3.2.6}$$

In Eq. 3.2.6, Δ_R^G is the average of the deviations of all n aggregate constraints.

Example 3.1. *Based on Example 1.1, the scientist might use the following query to find the desired sky region:*

```
SELECT * FROM SDSS.Star WHERE ( ra ≥ 179.5 and ra ≤ 182.3 )
and ( dec ≥ 1.24 and dec ≤ 1.86 )
WITH_CONSTRAINTS
SIMILARITY α = 0.5 AND
DISTANCE_FUNCTION = L1-norm AND
COUNT(*) = 1000 AND
AVG(brightness) = 2.3;
```

Clearly, one can see that her two aggregate constraints are 1000 objects and average brightness equals 2.3, respectively. Also, she prefers a result that satisfies the aggregate and similarity constraints equally (i.e., $\alpha = 0.5$). ■

In the following sections, we present our contributions towards the Similarity-aware, Aggregate-based Query Refinement problem defined in Def. 3.1. First, in Section 3.3 we discuss a special case of the Similarity-aware, Aggregate-based Query Refinement problem, where the aggregate constraint is the cardinality of a query's result. Then, in Section 3.4 we address the general case of this problem where aggregate constraints can be generated using the standard SQL aggregate functions. We also demonstrate the applicability of our proposed techniques for the Similarity-aware, Aggregate-based Query Refinement problem in Section 3.5.

3.3 SAQR Schemes

Query Refinement techniques enable databases users to automatically adjust their queries so that the queries results satisfy some user-defined constraints. Setting a constraint on the *cardinality* of the query result is one example of such constraints, which provides practical solutions to the problem of queries returning too many or too few answers, and has recently attracted several research efforts (e.g., [79, 14, 123, 78]).

Towards this, it has been shown that simple *local search* techniques based on greedy heuristics (e.g., *Hill Climbing*) often provide efficient and effective solutions to the cardinality-based query refinement problem [14]. In particular, the relationship between the possible refined queries and their corresponding cardinalities exhibit a *monotonic* pattern [14], which makes it practical to apply local search methods.

These techniques, however, are oblivious to the (dis)similarity between the input query and its corresponding refined version. That is, to meet the cardinality constraint, the generated refined query might often be very far (i.e., dissimilar) from the input query. While the (dis)similarity between two queries can be quantified using several alternative measures (e.g., [118, 98, 57, 90, 123]), it should be clear that, irrespective of the adopted similarity measure, a refined query that is very different from the input one will have a very limited benefit to the end user and is often rendered useless.

To address the limitation of current cardinality-based QR techniques, we propose to include users' preferences in refinement, in which the user satisfaction is measured in terms of both: 1) meeting some specified cardinality constraint on the refined query, and 2) maximizing the similarity between the submitted input query and its corresponding refined one. Achieving such goal is rather a challenging task as it would require an exhaustive examination of the large space of possible query refinements. In particular, the monotonic property exhibited by the cardinality constraints no longer holds for our dual-criteria objective, in which cardinality is combined with similarity. Hence, current QR techniques that are based on local search heuristics have higher chances of meeting a local minima and falling short in finding a refined query that strikes a fine balance between minimizing the deviation in cardinality and maximizing the similarity.

Motivated by that challenge, we propose a novel scheme called Similarity-aware Query Refinement (SAQR), which aims to balance the tradeoff between satisfying the cardinality and similarity constraints imposed on the refined query so that to maximize its overall benefit to the user.

In the following subsections, we formally define the special case of Similarity-aware, Aggregate-based Query Refinement problem, which captures the user's constraints on both cardinality and similarity. Then, we present innovative schemes called SAQR, which utilizes pruning techniques

based on both similarity and cardinality to efficiently formulate a refined query that meets the user's constraints. Further, we show how SAQR employs a hierarchical representation of the refined queries search space, which allows for significant reduction in the cost incurred by SAQR while maintaining the quality of its solution. We demonstrate the consistent, significant gains provided by SAQR schemes when compared to existing QR techniques by conducting extensive experiments on the TPC-D benchmark database.

3.3.1 Problem Statement

As mentioned above, setting a cardinality constraint on a query's result is a special case of the problem defined in Def. 3.1. Formally:

Definition 3.2. *Given a database B, an input conjunctive query I, a distance function $D()$, and a cardinality constraint K over the result $I(B)$, the goal is to find R that satisfies the cardinality constraint K while minimizing $D(R,I)$.* ■

Given the cardinality constraint K, we rewrite Eq. 3.2.4 and Eq.3.2.5 to be as follows:

$$\Delta_R^K = \frac{1}{z}|K - K_R| \tag{3.3.1}$$

$$\Delta_R = \alpha\Delta_R^S + (1-\alpha)\Delta_R^K \tag{3.3.2}$$

where the cardinality constraint K is the corresponding aggregate constraint G defined in Def. 3.1, $K =$`count(*)`, and Δ_R^K is the deviation in cardinality. Hence, the goal is to find R that minimizes Δ_R.

Next, we introduce our two algorithms SAQR-S and SAQR-CS that aim to efficiently and effectively achieve that goal. We discuss how the search space is represented, the optimization techniques used by these two algorithms, and the properties and techniques that are used by our algorithms in optimizing the search.

3.3.2 SAQR-S

In this section, we present our SAQR-Similarity scheme (SAQR-S for short, outlined in Algorithm 3.1.), which leverages the distance constraint to effectively prune the search space. Then, in the next section, we present SAQR-CS, which extends SAQR-S by exploiting the cardinality constraint for further pruning of the search space and higher efficiency.

Our similarity-aware query refinement problem, as defined in Eq.3.3.2, is clearly a preference query over the query space $\mathbb{R}$ and naturally lends itself as a special instance of a *Top-K* or *Skyline*

Algorithm 3.1 SAQR-S

```
1:  Input: Input Query I, K, α, δ
2:  Output: Refined Query R_opt, Δ_min.
3:  K_I = getCardinality(I);
4:  Δ_I^S = 0.0; Δ_I = αΔ_I^S + (1 − α)Δ_I^G;
5:  Δ_min = Δ_I; Δ_TA = 0.0;
6:  R_c ← GenerateQueries(I,δ);
7:  while ( R_c ≠ ϕ ) do
8:      for ( all R_i in R_c ) do
9:          Δ_TA = αΔ_{R_i}^S + (1 − α) × 0;          ▷ Compute threshold deviation for R_i
10:         if ( Δ_TA < Δ_min ) then
11:             K_{R_i} = getCardinality(R_i);          ▷ Call database layer
12:             Δ_{R_i} = αΔ_{R_i}^S + (1 − α)Δ_{R_i}^K;
13:             if (Δ_{R_i} < Δ_min) then          ▷ If R_i has better deviation than best
14:                 Δ_min = Δ_{R_i}; R_opt = R_i;          ▷ Store best deviation and its query
15:         else
16:             stop;          ▷ No ungenerated query will have deviation < Δ_min
17:     R_c ← GenerateQueries(R_c,δ);
18: return R_opt, Δ_min
```

queries. In particular, our goal is to search the query space $\mathbb{R}$ for the one refined query R_{opt} that minimizes the objective function defined in Eq. 3.3.2.

Such query R_{opt} is equivalent to a Top-1 query over the total of two attributes: 1) similarity deviation (i.e., Δ_R^S), and 2) cardinality deviation (i.e., Δ_R^K). R_{opt} should also fall on the skyline of a 2-dimensional space over those two attributes [47, 116, 118]. However, efficient algorithms for preference query processing (e.g., [75, 32]), are not directly applicable to the similarity-aware refinement of aggregation queries problem, for the following reasons:

1. For any query $R_i \in \mathbb{R}$, the values of $\Delta_{R_i}^S$ and $\Delta_{R_i}^K$ are not physically stored and they are computed on demand depending upon the input query I and the specified cardinality constraint K.

2. In addition to the cardinality constraint K, computing $\Delta_{R_i}^K$ for any query $R_i \in \mathbb{R}$, requires an expensive probe to estimate the cardinality $K_{R_i} = |R_i(B)|$ of query R_i.

3. The size of the query search space $|\mathbb{R}|$ is prohibitively large and potentially infinite.

To address the limitations listed above, we propose the SAQR scheme for similarity-aware refinement of cardinality constraints. In particular, SAQR adapts and extends algorithms for Top-K query processing towards efficiently and effectively solving the problem of similarity-aware refinement with cardinality constraints. Before describing SAQR in details, we first outline a baseline solution based on simple extensions to the *Threshold Algorithm (TA)* [31].

Conceptually, to adapt the well-known TA to the query refinement model, each possible refined query $R_i \in \mathbb{R}$ is considered as an object with two *partial scores*: 1) partial score based on deviation in similarity (i.e., $\Delta^S_{R_i}$), and 2) partial score based on deviation in cardinality (i.e., $\Delta^K_{R_i}$). Those two partial scores are maintained in two separate lists: 1) Δ^S-list, and 2) Δ^K-list, which are sorted in a descending order based on their deviation.

Under the classical TA algorithm, the two sorted lists are accessed at parallel. When an object's partial score is retrieved from a list (i.e., either Δ^S-list or Δ^K-list) by a sorted access, its other partial score is also fetched from the other list by a random access and the object's score is kept in a buffer along with the object itself. A threshold value T is computed from the scores of the last seen objects *from the sorted access*.

TA reports any objects in the buffer with a score above T, and terminates when the lists are traversed to completeness or the number of required objects has been met.

Clearly, such straightforward conceptual implementation of TA is infeasible to the similarity-aware refinement of cardinality constraints problem due to the three reasons listed earlier. To address the first reason (i.e., absence of partial deviation values), SAQR-S generates the Δ^S-list on the fly and on-demand based on the input query I. In particular, given query I, it progressively populates the Δ^S-list with the distance between I and the nearest possible refined query $R_i \in \mathbb{R}$.

To control and minimize the size of the search space, a value δ is defined and the nearest query is defined in terms of that δ. In particular, given an input query I, a first set of nearest queries is generated by replacing each predicate $a_i \leq x_i^I$ with two predicates:

- $a_i \leq x_i^I + \delta$
- $a_i \leq x_i^I - \delta$

The same process is then repeated recursively for each set of generated queries.

Clearly, using δ allows for simply *discretizing* the rather continuous search space $\mathbb{R}$. Hence, $\mathbb{R}_\delta$ can be perceived as a uniform grid of granularity δ (i.e., each cell is of width δ). We note that at any point of time, the Δ^S-list is always sorted since the values in that list are generated based on proximity.

One approach for populating the Δ^K-list is to first generate the distance Δ^S-list and then compute the corresponding Δ_i^K value for each query R_i in the Δ^S-list. Those values are then sorted in descending order and the TA algorithm is directly applied on both lists. Clearly, that approach has the major drawback of probing the database for estimating the cardinality of all the possible queries in the new discretized search space. Instead, we leverage the particular *Sorted-Random (SR)* model of the Top-K algorithm to minimize the number of those expensive estimation probes.

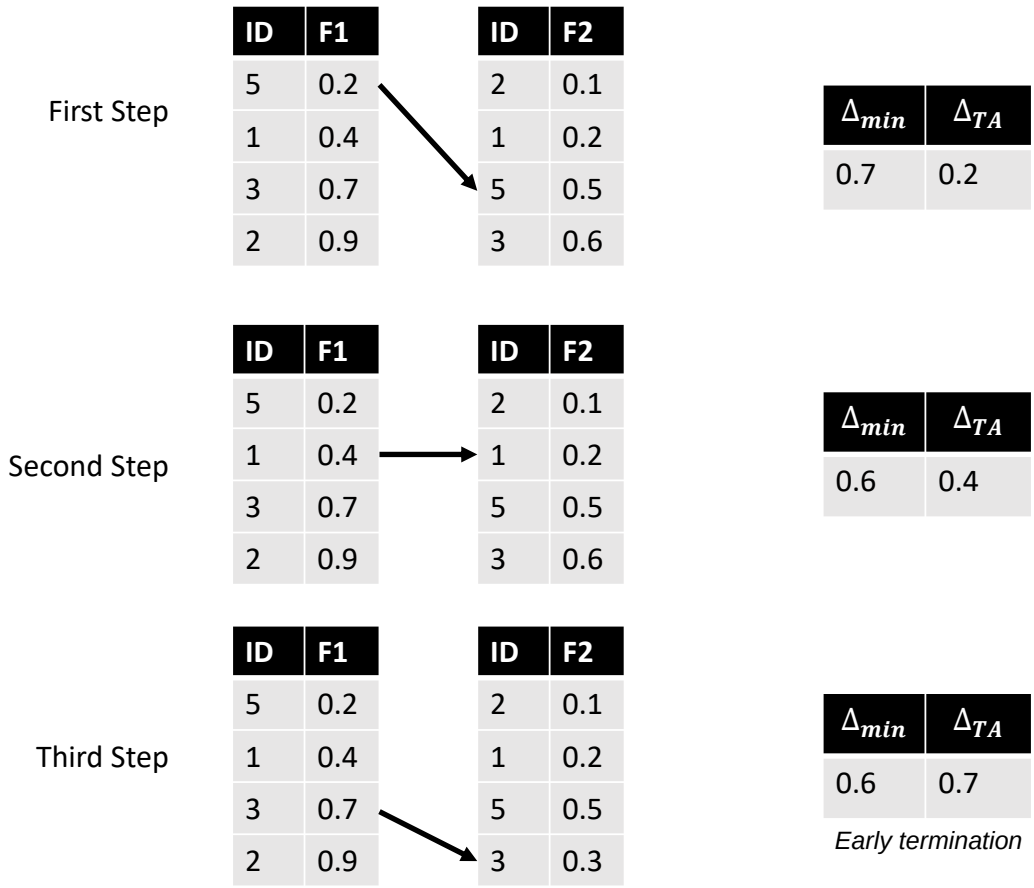

Figure 3.3: The first three steps of TA-Algorithm under the SR Model with two list: access sorted list(left) and random access list (right)

The SR model is particularly useful in the context of web-accessible external databases, in which one or more of the lists involved in an objective function can only be accessed in random and at a high-cost [75, 32, 47]. In that model, the sorted list (i.e., S) basically provides an initial set of candidates, whereas random lists (i.e., R) are probed on demand to get the remaining partial values of the objective function.

In our model, the Δ^S-list already provides that sorted sequential access, whereas Δ^K-list is clearly an external list that is accessed at the expensive cost of probing the database. Under that setting, while the Δ^S-list is generated incrementally, two threshold values are maintained (as in [75, 32]):

- Δ_{min}: The minimum calculated deviation Δ that have been found so far.
- Δ_{TA}: The minimum possible deviation Δ of a query that is yet to be estimated.

The two thresholds listed above enable efficient navigation of the search space by pruning a significant number of the queries in $\mathbb{R}_\delta$. This is achieved by means of a simple technique referred to as *Early Termination*. Early termination kicks in when a query R_i is generated, and assumed to have

zero cardinality deviation (Alg. 3.1, line 9), but its deviation threshold Δ_{TA} is higher than or equal to the best found query so far Δ_{min}.

Figure 3.3 illustrates the first three steps of the SR Model with two lists: left-hand-side list provides sorted access while the right-hand-side provides random access at high cost. Initially, the two thresholds Δ_{min} and Δ_{TA} are set to 1. For simplicity, assume that the goal is to find the query's ID with the minimum deviation, i.e., find the minimum Δ where $\Delta = F1 + F2$. In the third step, the algorithms finds out that query with ID 3 has a higher threshold Δ_{TA} than Δ_{min}. That is, the early termination condition is met. Hence, the algorithms stops the search and ignores the rest of candidate queries in the sorted list.

Theoretical Analysis: We next analyze the completeness and soundness of the algorithm SAQR-S presented above. Recall that for any arbitrary input query I and a cardinality constraint K, SAQR-S outputs the optimal refined query R_{opt} such that R_{opt} has the minimum possible deviation Δ_{min} among all candidate queries in the search space $\mathbb{R}_\delta$.

First, we prove SAQR-S convergences. The algorithm recursively generates candidate queries (starting from I), and progresses by adding the new generated candidate queries into R_c. At each iteration, a single candidate query R_i is evicted from R_c, then new candidate queries are generated by refining R_i and are added to R_c. As long as R_c is not empty, the algorithm will not terminate. However, the algorithm only adds candidate queries into R_c as long as they have not been visited before. Hence, R_c will eventually become empty and the algorithm will converge. Moreover, the algorithm will also converge once it finds a candidate query with a similarity deviation higher than the minimum deviation found so far, i.e., if Δ_{TA} is higher than Δ_{min} then there is no other candidate query which has a deviation less than Δ_{min} (since SAQR-S iterates over candidate queries in ascending order based on their similarity deviation).

Second, we prove that SAQR-S is complete, i.e., given an arbitrary input query and cardinality constraint, it outputs the optimal refined query R_{opt} such that R_{opt} has the minimum possible deviation Δ_{min} among all candidate queries in the search space $\mathbb{R}_\delta$. Given α, δ, K and an input query I, there must be at least one refined query $R \in \mathbb{R}_\delta$ in the δ approximated search space such that R has the minimum possible deviation among all possible queries in the δ approximated search space, i.e., $\exists R \in \mathbb{R}_\delta$ such that $R = R_{opt}$. Assuming that α and K are set to zero, then R_{opt} is essentially any refined query in $\mathbb{R}_\delta$ with zero cardinality (or the minimum cardinality). One such refined query is a query that is located at the bottom-left corner of the approximated search space. Let this query be R_{BL}. As SAQR-S progresses and iterates over candidate queries, Δ_{min} decreases. Since α is set to zero, this guarantee that SAQR-S will visit R_{BL} (no queries will be pruned using the Early Termination test), which has the minimum possible deviation. Hence, SAQR-S outputs complete and correct answer.

It is also trivial to show that SAQR-S is complete when α is higher than zero. Note that when α is higher than zero (e.g., $\alpha = 0.5$) and K is set to zero, R_{opt} is no longer the candidate query with zero cardinality. It is the one with the minimum cardinality deviation weighted by $(1-\alpha)$. Because SAQR-S will visit all candidate queries (except the ones with a similarity deviation higher than the best found deviation Δ_{min}) then this guarantee that SAQR-S will come across the optimal refined query with the minimum cardinality deviation weighted by $(1-\alpha)$.

Complexity Analysis: We analyze next the complexity of SAQR-S. In the worst case, SAQR-S will visit all candidate queries in $\mathbb{R}_\delta$. The number of candidate queries in $\mathbb{R}_\delta$ depends on the number of predicates d and δ. Formally:

$$|\mathbb{R}_\delta| = (\frac{1}{\delta})_1 \times (\frac{1}{\delta})_2 \times ... \times (\frac{1}{\delta})_d = (\frac{1}{\delta})^d$$

3.3.3 SAQR-CS

The SAQR-S scheme, presented in the previous section, basically leverages the deviation in distance in order to bound the search space. Thus, it reduces the number of candidate refined queries to be generated, and in turn, reduces the number of probes needed for cardinality estimation. The underlying premise is that the optimal refined query R_{opt} is expected to be near the input query I. Hence, the thresholds from the TA algorithm effectively represent cutoff points after which no further refined queries need to be examined.

The SAQR-S scheme, however, still has two major drawbacks:

- It probes the database for estimating the cardinality for every candidate query R_i that survives the early termination test, and
- The overall search space is still large despite of the discretization process.

In this section, we propose the extended SAQR-Cardinality/Similarity scheme (SAQR-CS for short). This scheme is shown in Algorithm 3.2. At a high-level, SAQR-CS provides the following features:

1. SAQR-CS exploits the *monotonicity* property of the cardinality constraint so as to provide significant reductions in the search space, and
2. SAQR-CS employs a *hierarchical* representation of the search space that allows for adaptive navigation and further reductions in the total cost.

In the following, we describe in details the two features listed above.

Algorithm 3.2 SAQR-CS

```
1:  Input: Input Query I, K, α, δ
2:  Output: Refined Query R, Δ_min
3:  K_I = getCardinality(I);
4:  Δ_I^S = 0.0; Δ_I = αΔ_I^S + (1 − α)Δ_I^K;
5:  Δ_min = Δ_I; Δ_TA = 0.0;
6:  H = δ; h = 0.5;
7:  R_c ← GenerateQueries(I,h);
8:  while (h ≥ H) do
9:      while ( R_c ≠ φ ) do
10:         for ( all R_i in R_c ) do
11:             Δ_TA = αΔ_{R_i}^S + (1 − α) × 0;          ▷ Compute threshold deviation for R_i
12:             if ( Δ_TA < Δ_min ) then
13:                 K_i^u = getUpperBound(R_i);           ▷ cardinality of closest query dominating R_i
14:                 K_i^l = getLowerBound(R_i);           ▷ cardinality of closest query dominated by R_i
15:                 Δ_{R_i}^K = estCardDev(K_i^u, K_i^l, K);   ▷ Estimate cardinality deviation
16:                 Δ_{R_i} = αΔ_{R_i}^S + (1 − α)Δ_{R_i}^K;
17:                 if ( Δ_{R_i} < Δ_min ) then
18:                     K_{R_i} = getCardinality(R_i);    ▷ Call database layer
19:                     Δ_{R_i} = αΔ_{R_i}^S + (1 − α)Δ_{R_i}^K;   ▷ Compute exact deviation
20:                     if (Δ_{R_i} < Δ_min) then
21:                         Δ_min = Δ_{R_i}; R_opt = R_i;
22:             else
23:                 stop;                                 ▷ No ungenerated query will have deviation < Δ_min
24:         R_c ← GenerateQueries(R_c,h);
25:     clear(R_c); h_prev = h; h = h/2;                  ▷ Compute next's level granularity
26:     for ( each cell C in h_prev ) do
27:         R_i ← C;                                      ▷ Each cell represents a query
28:         R_c += GenerateQueries(R_i,h);                ▷ Generate queries for next level
29: Return R_opt, Δ_min;
```

3.3.4 The Monotonicity Property

Consider a candidate query R with d conjunctive range predicates $P_1 \wedge P_2 \wedge ... \wedge P_d$, such that each predicate P_i is defined as: $a_i \leq x_i^R$, for $i \neq j$, (similar to our query model presented in Section 3.2). Further, assume that n_i is the number of distinct values for attribute a_i.

The space of possible cardinalities of query R can be modeled as d-dimensional $n_1 \times ... \times n_d$ grid $\mathscr{G}$ [14]. The value of $\mathscr{G}[x_1,...,x_d]$ for $1 \leq x_i \leq n_i$ is precisely the cardinality of the query R when each predicate P_i is instantiated with the x_i-th smallest distinct value of attribute a_i. Therefore, $\mathscr{G}$ satisfies the following monotonicity property: $\mathscr{G}[x_1,...,x_d] \leq \mathscr{G}[y_1,...,y_d]$ when $x_i \leq y_i$ for every attribute a_i [14].

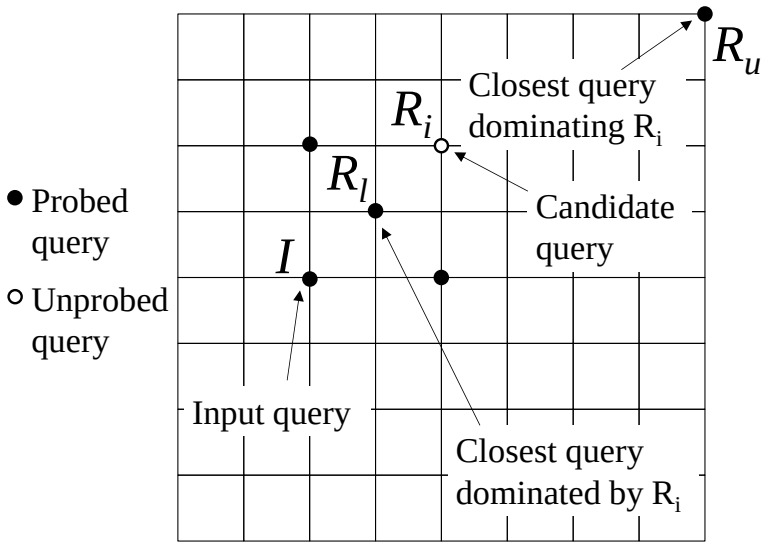

Figure 3.4: Estimating upper and lower bounds of Δ_{R_i} by using probed queries R_l and R_u.

3.3.5 Cardinality-based Pruning

SAQR-CS exploits the monotonicity property of the cardinality constraint so that to provide significant reductions in the search space. In particular, if a candidate refined query R_i passed the early termination test, SAQR-CS estimates a lower bound K_i^l and an upper bound K_i^u on the cardinality of query R_i (i.e., $K_i^l \leq K_i \leq K_i^u$).

Estimating those bounds is very efficient since it is completely based on the candidate queries that have been examined so far and thus requires no probing of the database.

SAQR-CS exploits the monotonicity property as follows: it keeps track of the queries that have been generated and examined while progressively populating the Δ^S-list. Then, when a new query R_i is generated, SAQR-CS sets the bounds K_i^l and K_i^u as follows (See Figure 3.4):

- $K_i^l = K_l$, where K_l is the cardinality of query R_l, which is the closest query *dominated* by R_i. That is, when $x_j^l \leq x_j^i$ for every attribute a_j.

- $K_i^u = K_u$, where K_u is the aggregated value of query R_u, which is the closest query *dominating* R_i. That is, when $x_j^i \leq x_j^u$ for every attribute a_j.

After finding the bounds K_i^l and K_i^u on the value of K_i, SAQR-CS then assesses the benefit of probing the database to get an accurate estimate for the deviation of R_i. In particular, SAQR-CS estimates the deviation of R_i given the possible range of the cardinality value $[K_i^l - K_i^u]$ it might have. Thus, it is required to test if any value in that range can provide an overall deviation that is smaller than the deviation achieved so far (i.e., $\Delta_{R_i} < \Delta_{min}$).

To perform that test, SAQR-CS initially retrieves the distance between R_i and I, i.e., $\Delta_{R_i}^S$. Then,

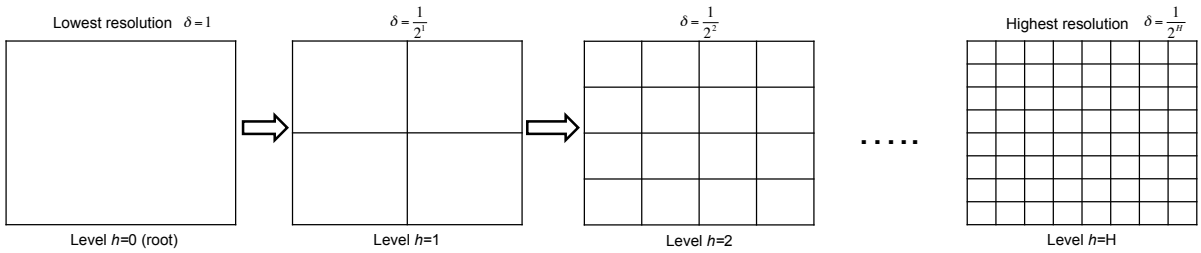

Figure 3.5: 2-Dimensional search space is decomposed into H levels, where the resolution of the top level $\delta = 1$, and the resolution of the bottom level H is $\delta = \frac{1}{2^H}$

it calculates the minimum possible cardinality deviation $\Delta^K_{R_i}$ by *estCardDev()* using K^l_i and K^u_i, as shown in Alg. 3.2, line 15.

The minimum possible cardinality deviation $\Delta^K_{R_i}$ is computed by Eq. 3.3.1 depending upon where the cardinality constraint K lays within the range $[K^l_i - K^u_i]$. Formally:

$$\Delta^K_{R_i} = \begin{cases} \frac{|K - K^l_i|}{z} & \text{if } K < K^l_i \\ \frac{|K - K^u_i|}{z} & \text{if } K > K^u_i \\ 0 & \text{Otherwise} \end{cases} \tag{3.3.3}$$

Finally, SAQR-CS substitutes $\Delta^K_{R_i}$ and $\Delta^S_{R_i}$ in Eq. 3.3.2 to estimate Δ_{R_i} (Alg. 3.2 line 16). If Δ_{R_i} is less than Δ_{min}, then R_i might provide a smaller deviation and the database is probed to retrieve its actual cardinality (Alg. 3.2 line 18).

3.3.6 Hierarchical Representation of the Search Space

Clearly, the effectiveness of the bounds described above on pruning the search space depends on the tightness of the cardinality bounds K^l_i and K^u_i. However, achieving such tight bounds is not always possible when the candidate refined queries are generated in order of their proximity to the input query I on a uniform grid with a constant width δ such as the one described in the previous section.

For instance, under that approach, a generated candidate query R_i that is positioned between the input query I and the origin for the search space, will often have a loose lower bound K^l_i. Similarly, if R_i is positioned between I and the limits of the search space, then it will have a loose upper bound of K^u_i.

To achieve tighter bounds, SAQR-CS employs a hierarchical representation of the search space based on the *pyramid* structure [4, 67] (equivalent to a partial quad-tree [68, 35]). The pyramid decomposes the space into H levels (i.e., pyramid height). For a given level h, the space is partitioned into 2^{dh} equal area d-dimensional grid cells. For example, at the pyramid root (level 0), one cell

represents the entire search space, level 1 partitions space into four equal-area cells, and so forth.

To create the pyramid representation, SAQR-CS generates candidate queries recursively in iterations using a dynamic δ. In particular, the value of δ_h in any iteration h is equal to $\frac{1}{2^h}$ (see Figure 3.5). The queries generated in iteration i are processed similar to SAQR-CS (as described in the previous section). This is in addition to: 1) applying the cardinality-based pruning outlined above, and 2) maintaining the minimum deviation Δ_{min} across iterations.

The pyramid representation provides the following advantages:

- Effective pruning: the pyramid representation allows SAQR-CS to compute the cardinality bounds K_i^l and K_i^u for a candidate query R_i based on already probed queries that are either at the same level or higher levels in the pyramid. This provides better coverage of the search space and tighter bounds.

- Efficient search: the pyramid representation allows SAQR-CS to quickly zoom-in to the area where R_{opt} is located.

To understand how SAQR-CS employs the pyramid representation to achieve effective and efficient search, assume R_{opt} is located somewhere in the third quarter (top-right) of the highest-resolution grid in Figure 3.5, while the input query I is located at the first quarter (bottom-left). Indeed, the pyramid representation allows SAQR-CS to jump quickly to where R_{opt} is located, since it visits the search space level by level, from the lowest resolution to the highest.

Unlike SAQR-CS, SAQR-S sees the search space as one level in the highest resolution, and it cannot jump to R_{opt} unless it visits all queries between R_{opt} and I. However, when R_{opt} is located next to I (e.g., both of them are in the first quarter), SAQR-S will reach the optimal solution earlier than SAQR-CS, because it takes SAQR-S only few steps to find it while SAQR-CS will have to go through multiple steps, going from the top level of the pyramid to the bottom.

3.3.7 Experiments

Firstly, we present the experimental setup then we discuss the findings and results. The experiments settings are summarized in Table 3.3.

Schemes: We have experimented with the following algorithms:

- **Hill Climbing (HC):** This is the scheme proposed in [14] to automatically generate queries with cardinality constraints for DBMS testing. HC discretizes the search space and navigates it in a greedy manner until no further reduction in deviation is attainable. However, in this work

Parameter	Range	Default
Similarity weight (α)	0.0–1.0	0.5
Dimensions (d)	1–4	2
Grid resolution (δ)	$1/2^5$–$1/2$	$1/2^5$
Database Size ($\|B\|$)	60k, 600k, 3M, 6M tuples	60k tuples

Table 3.3: Evaluation Parameters.

we have extended the HC scheme proposed in [14] use our similarity-aware objective function (i.e., Eq. 3.3.2).

- **SAQR-S:** Our proposed scheme, which utilizes similarity for navigating and pruning the search space (as described in Section 3.3.2).
- **SAQR-CS:** Our proposed scheme, which extends SAQR-S and utilizes both similarity and cardinality for navigating and pruning the search space (as described in Section 3.3.3).

All algorithms were implemented as a Java front-end on top of the MySQL DBMS.

Datasets: In our experiments, we use real TPC-D datasets of different scales. The dataset is created similar to [79], using the publicly available tool [19] which provides the capability of generating datasets with different scales. In our experiments, all the numerical columns in the generated tables are normalized in the range [0-1].

Queries: To cover a large spectrum of query contraction and expansion scenarios, we generated a set of 100 <query, cardinality> pairs. Each pair is an input query together with its cardinality constraint generated according to a uniform distribution.

Performance Measures: Performance is evaluated in terms of:

- **Average cost (C_R):** That is the average number of probes (calls) made to the database *evaluation layer* for refining all the queries in the workload.
- **Average deviation (Δ_R):** That is the average deviation experienced by all the queries in the workload, where the deviation perceived by each query is computed according to Eq. 3.3.2.

Impact of Similarity Weight

In Figures 3.6 and 3.7, we use the default experiment settings while varying the similarity weight α from 0.0 to 1.0.

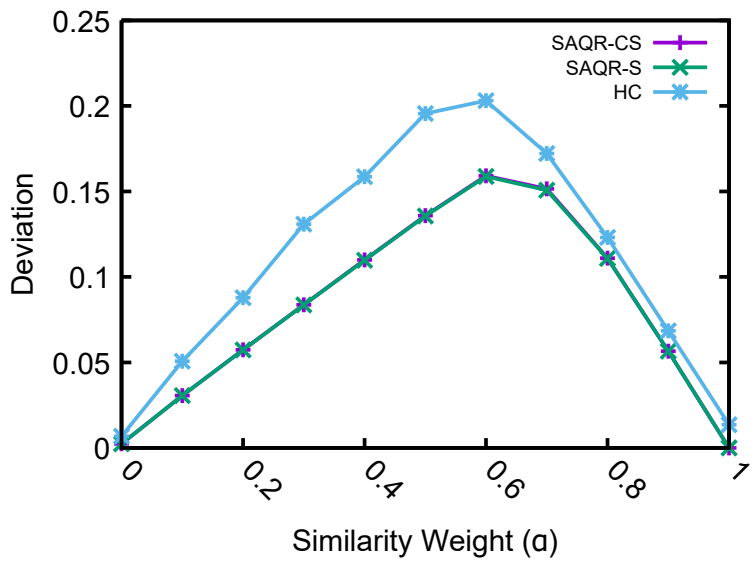

Figure 3.6: Average deviation while varying similarity weight α

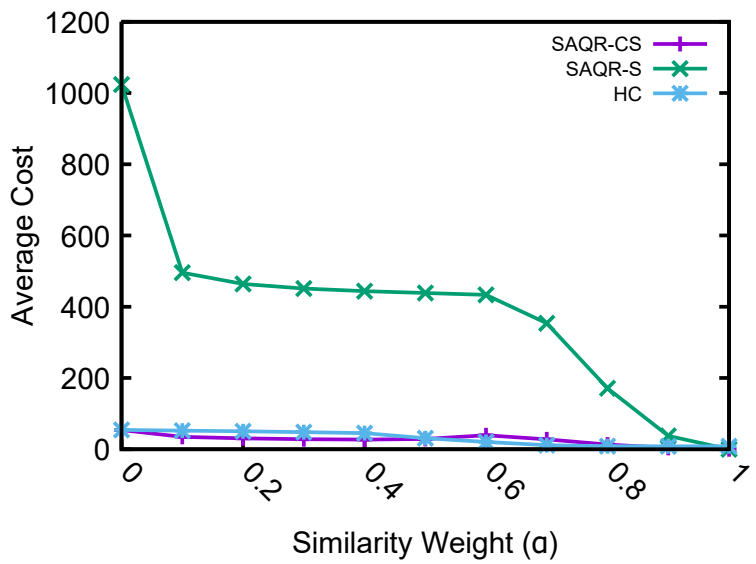

Figure 3.7: Average cost while varying similarity weight α

Figure 3.6 shows that all schemes follow the same pattern: deviation increases with α until it reaches a critical point after which it rebounds. The reason behind this pattern is that minimizing deviation in cardinality while maximizing similarity are two objectives that are typically in conflict. The degree of conflict is determined by α, for example at $\alpha = 1$ there is no conflict and the total deviation is based on similarity. Hence, SAQR easily finds a refined query that satisfies K, though it might be very dissimilar from the input. For α around 0.5, both objectives are of equal importance, making it hard to find a refined query that satisfies them both simultaneously leading to higher deviation (compared to $\alpha = 1$). However, SAQR effectively balances that tradeoff and finds a near-optimal solution. For instance, when α equals 0.5, SAQR-CS achieves 30% better deviation than HC.

Figure 3.7 shows the high efficiency of both HC and SAQR-CS compared to SAQR-S. However, at higher values of α, the cost of SAQR-S is relatively low. This is because SAQR-S traverses the grid cells starting from the closest point to the input query (smallest distance) to the furthest one. Thus, assigning higher weight to deviation in similarity increases the chance for SAQR-S to satisfy the early termination condition.

Setting the value of α depends on the user preference of similar refined queries to her initial query. As a guideline, setting α to zero provides refined queries that satisfy the cardinality constraint regardless of the similarity to the input query. As the similarity weight is increased from zero to one, the resultant refined queries are more similar the input queries, however, this introduces a conflict on satisfying the cardinality constraint, which is apparent in the experiment above.

Impact of Grid Resolution

In this experiment, we examine the impact of the grid resolution δ on the performance metrics.

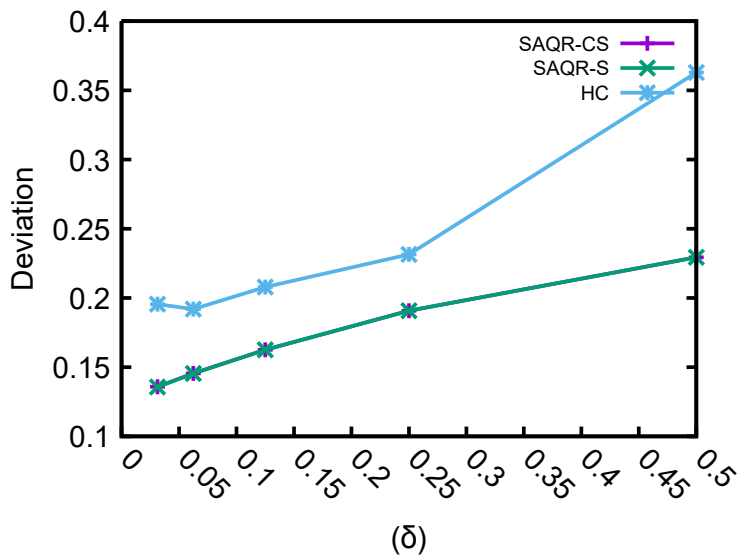

Figure 3.8: Average deviation while varying grid resolution δ

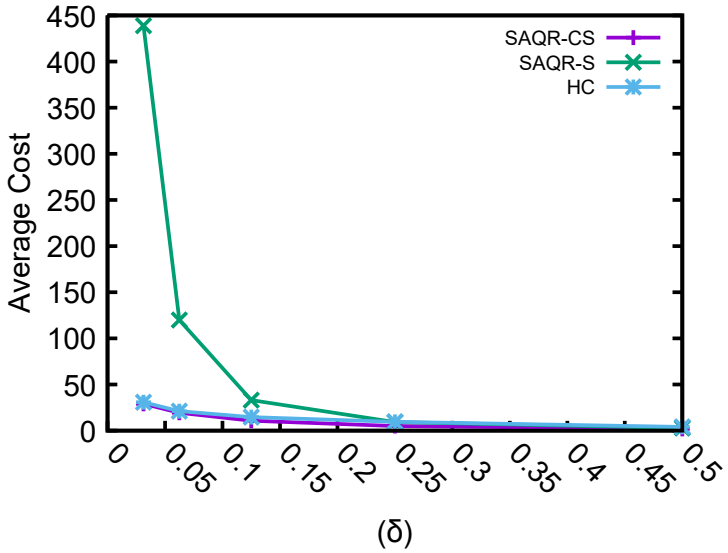

Figure 3.9: Average cost while varying grid resolution δ

As shown in Figure 3.8, all three schemes exhibit a direct correlation between δ and the average deviation. This relation is natural because when δ is increased, the search space is highly approximated, which essentially increases the probability of missing the exact target constraints, and vice versa. Figure 3.9 shows that the average cost drops as δ is increased. For instance, the average cost provided by SAQR-S is reduced by 72% when δ increased from $(\frac{1}{2})^5$ to $(\frac{1}{2})^4$, while the reduction in the costs of HC and SAQR-CS is 31% and 33%, respectively. This is because the total number of cells in the grid is decreased, leading to less number of cells to be scanned. Looking at both Figures 3.8 and 3.9 uncovers the trade-off between the deviation and cost metrics which is controlled by δ.

Specifically, users should tune δ to control the trade-off between the deviation and cost such that lowering the value of δ provides more accurate result at the expense of more cost.

Impact of Dimensionality

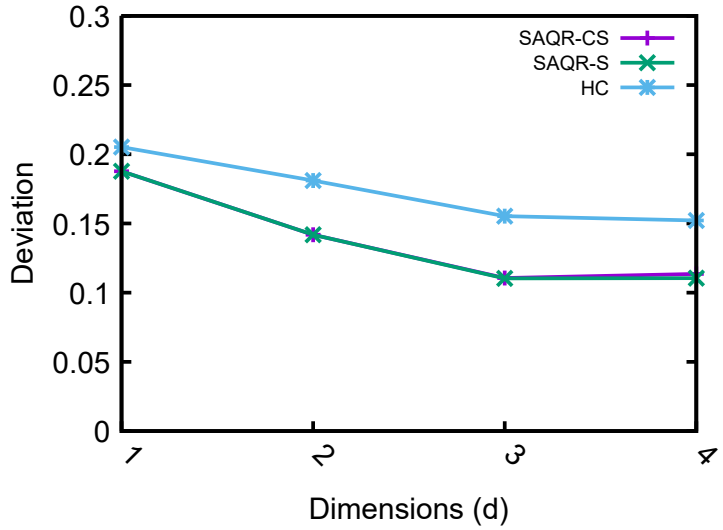

Figure 3.10: Average deviation while varying number of dimensions d

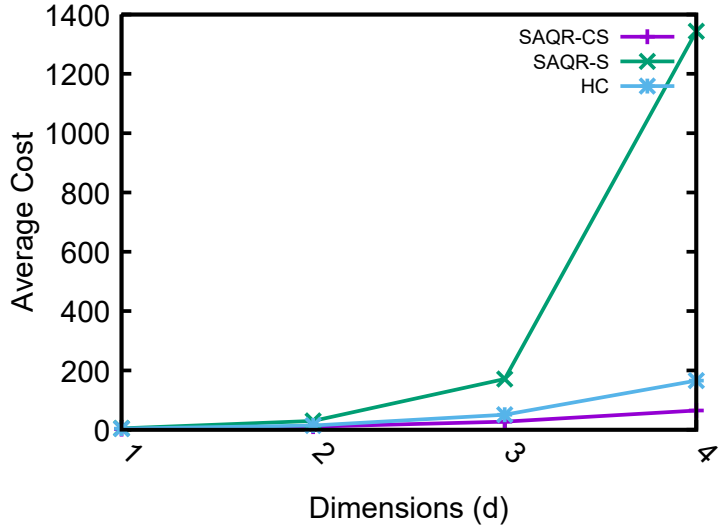

Figure 3.11: Average cost while varying number of dimensions d

In this experiment we measure the effect of dimensionality d on performance, while $\delta = (\frac{1}{2})^3$. Figure 3.10 shows that SAQR achieves better deviation than HC for all values of d. For instance, when all queries include four dimensions, SAQR-CS and SAQR-S reduced the deviation by 28% when compared to HC. The reason for the poor performance of HC is that when the number of dimensions is increased, there are more possible refined queries, which increases the chances for HC to get stuck at local minima and miss the optimal solution. Figure 3.11 shows that, in general, SAQR-CS outperforms all other schemes. For instance, for four-dimensional queries, SAQR-CS reduced the cost compared to HC and SAQR-S by 60% and 95%, respectively. Clearly, these reductions are due to the cardinality-based pruning and pyramid structure techniques employed by SAQR-CS.

Impact of Database Size

Database Size	Probing time (ms)
60K	21
600K	338
3M	1647
6M	3237

Table 3.4: Time per probe in milliseconds for TPC-D database in different scales

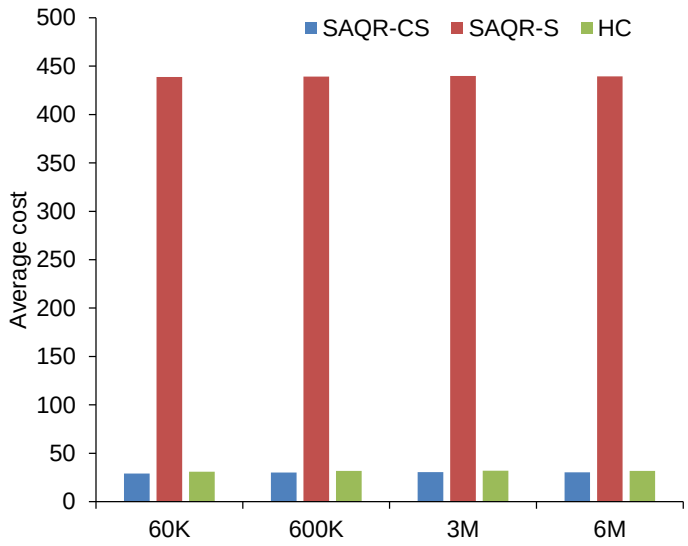

Figure 3.12: Average cost for TPC-D database in different scales

In this experiment, we generated four scaled versions of the TPC-D database of sizes 60K, 600K, 3M, and 6M tuples. As expected, Figure 3.12 shows the number of probes for each scheme is constant for all four databases. Clearly, this is because the search space of the refined queries remains the same for all sizes. However, as it is also expected, the database size determines the amount of data processed in each probe and in turn, the probing time (as shown in Table 3.4). In particular, for a machine loaded with Intel Core i7 3.40GHz CPU, 16.0 GB RAM, and Windows 7 OS, Table 3.4 shows the time per probe for the different database sizes. Combining the results in Figure 3.12 and Table 3.4 shows that SAQR-CS allows for scalable and practical query refinement.

Value range \ z-value	z=0	z=1	z=2	z=3
0.0 – 0.2	19%	21%	10%	2%
0.2 – 0.4	20%	23%	23%	14%
0.4 – 0.6	20%	16%	2%	0%
0.6 – 0.8	20%	23%	61%	83%
0.8 – 1.0	21%	17%	3%	0%

Table 3.5: A histogram of the data distribution of different z-value for `quantity` attribute in the `lineitem` table. z=0 represents a uniform distribution, while z=3 represents a highly skewed distribution

Impact of Data Skewness

In this experiment we report the impact of data skewness on our proposed algorithms. We generated skewed data using the publicly available tool [19], which provides the capability of generating TPC-D datasets with different z distributions. Table 3.5 shows a histogram of skewed distribution of the `quantity` attribute in the `lineitem`. Note that $z = 0$ represents a uniform distribution, while $z = 3$ represents a highly skewed distribution.

For this particular experiment, we set $\alpha = 0$ to study the impact of data skewness on SAQR-CS pruning power. The remaining parameters are set to default. Also, we experimented with four z-value skewed distributions: 0, 1, 2 and 3.

As shown in Figure 3.13, the average number of probed queries varies with different z-values. In particular, the cardinality-based pruning technique implemented by SAQR-CS benefits from skewed data since the estimated cardinality bounds becomes tighter when data exhibits skewness. However, as Figure 3.14 shows, the deviation increases when data exhibit skewness. In highly skewed data, it is very difficult for any algorithm to find an optimal query that achieves the cardinality constraint. Specifically, when expanding or contracting a predicate of a candidate query over skewed data, the cardinality of this query will significantly change. Such significant changes in cardinality prevents any algorithm from getting very close to the cardinality constraint. Alternatively, expanding or contracting a predicate of a candidate query over uniformally distributed data will slightly change the cardinality of the query, hence, there is a high chance to achieve the cardinality constraint.

Results Discussion

Overall, SAQR-CS outperforms SAQR-S and HC in terms of deviation and cost, due to its cardinality-based pruning technique. Increasing the grid resolution δ results in high costs but lower

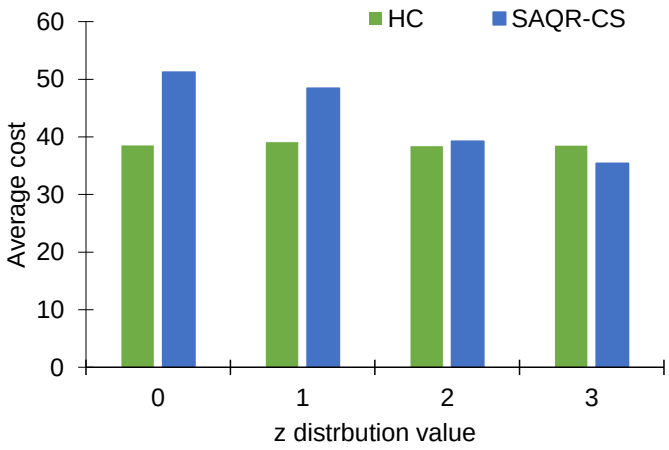

Figure 3.13: Average cost while varying z-value

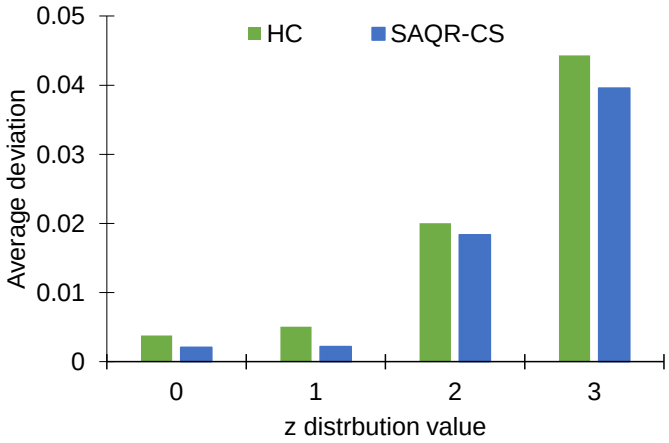

Figure 3.14: Average deviation while varying z-value

deviation for all algorithms. The number of probed queries for all algorithms remains the same for different dataset sizes because the search space is determined by δ and the number of dimensions d. Skewed data slightly increases SAQR-CS pruning power, however, the average deviation increases since it becomes more difficult to find a refined query that meets the cardinality constraint.

3.4 EAGER Schemes

In this section we present our contribution towards the Similarity-aware, Aggregate-based Query Refinement problem formally defined in Section 3.2.3:

Definition 3.3. ***Similarity-aware, Aggregate-based Query Refinement problem:*** *Given a database B, an input conjunctive query I, a distance function $D()$, and an aggregate constraint G over the result $I(B)$, the goal in the Similarity-aware, Aggregate-based Query Refinement problem is to find R that satisfies the aggregate constraint G while minimizing $D(R,I)$.* ■

Note that this problem preserves the hardness of the special case addressed earlier in Section 3.3.1. Specifically, all aggregate constraints addressed in the above problem follow the same monotonic property of cardinality. Using a Hill Climbing approach to search for an optimal query (given an aggregate constraint) will return sub-optimal solutions when similarity is included in the objective function.

Our proposed efficient refinement of aggregates constraints schemes (EAGER) extend SAQR schemes (Section 3.3) to cater for aggregate constraints. Similar to SAQR-S, EAGER-S employs the similarity constraint to prune the search space while EAGER-GS extends SAQR-CS and likewise employs the similarity and aggregate constraints to achieve efficient query refinement. We present approximation and optimization techniques that EAGER-GS employs as well to provide efficient and effective solutions for the Similarity-aware, Aggregate-based Query Refinement problem.

First, we investigate the monotonicity property of the aggregate operators `sum, avg, min, max`, which EAGER utilizes to compute lower and upper bounds of the aggregate constraints. We also provide a workaround for the aggregate `avg()` since it is not a monotonic function.

3.4.1 Aggregates Constrains Bounds

The monotonicity property introduced in Section 3.3.4 for computing lower and upper bounds for the cardinality constraints can also be used for the aggregate constraints `sum, avg, min, max`. Similar to SAQR, these bounds enable EAGER to prune unqualified candidate queries without probing the database to estimate their aggregate values. Specifically, when a candidate query R_i is generated, EAGER sets the aggregate bounds G_i^l and G_i^u as follows:

- $G_i^l = G_l$, where G_l is the aggregate value of query R_l, which is the closest query *dominated* by R_i. That is, when $x_j^l \leq x_j^i$ for every attribute a_j.
- $G_i^u = G_u$, where G_u is the aggregate value of query R_u, which is the closest query *dominating* R_i. That is, when $x_j^i \leq x_j^u$ for every attribute a_j.

However, the operator $\texttt{avg}(a_i)$ is a special case, as the monotonicity property does not hold for average. Hence, G_i^l and G_i^u for $\texttt{avg}(a_i)$ are estimated differently than the other aggregate operators.

To find G_i^l and G_i^u for $\texttt{avg}(a_i)$, the average of each probed query R_j is stored as $\texttt{count}(a_i)$ and $\texttt{sum}(a_i)$. Then, the lowest average of R_i is computed as if R_i has the $\texttt{count}(a_i)$ of the upper bound, and the $\texttt{sum}(a_i)$ of the lower bound. Analogously, the highest average of R_i is computed as if R_i has the $\texttt{count}(a_i)$ of the upper bound, and the $\texttt{sum}(a_i)$ of the lower bound plus the maximum value of a_i times the difference between the upper and lower bound $\texttt{count}(a_i)$. Formally:

$$G_i^l = \frac{G_l.\texttt{sum}(a_i)}{G_u.\texttt{count}(a_i)} \tag{3.4.1}$$

$$G_i^u = \frac{G_l.\texttt{sum}(a_i) + V}{G_u.\texttt{count}(a_i)} \tag{3.4.2}$$

Where V is the difference between G_u and G_l $\texttt{count}(a_i)$ aggregate.

After finding the bounds G_i^l and G_i^u on the value of G_i, EAGER (similar to SAQR) assesses the benefit of probing the database to get an accurate estimate for the deviation of R_i.

Multiple Aggregate Constraints: In practice, EAGER schemes can be easily extended to support multiple aggregate constraints. This can be achieved by replacing the deviation Δ_R^G in Eq. 3.2.4 with Eq. 3.2.6. Further, EAGER-GS can avoid probing the database layer for an aggregate G if its lower and upper bounds are exactly the same, i.e., if $G^u = G^l$. For ease of presentation and simplicity, we consider only one aggregate constraint in the rest of this chapter.

3.4.2 Optimization Techniques

While EAGER is all about reducing the cost without effecting the deviation, still, there is a need for more cost reductions. The reason is, in an interactive context, users expect to see results instantly. Hence, in the next section, we describe optimization and approximation techniques to increase the efficiency of our scheme EAGER. We propose an optimization technique to reduce the cost of refinement with a small footprint of storage and processing. Specifically, EAGER scheme materializes a set of selected candidate queries before exploring the search space to utilize them efficiently. Also, we propose approximation techniques for EAGER to improve the user experience with the scheme at an acceptable level of accuracy.

Level-based Materialization

A straightforward method to achieve less cost is to materialize candidate queries in the search space before running the schemes. Hence, when a candidate query is generated, if it happens to be

materialized, then there is no need to call the database layer to retrieve its aggregate.

We have utilized this method in EAGER scheme by materializing candidate queries based on levels chosen by the user. As shown previously in Section 3.3.6, the search space traversed by EAGER scheme is represented as a hierarchical structure. This structure consists of multiple levels, as illustrated in Figure 3.5. Users can specify which level(s) are to be materialized beforehand. Consequently, all candidate queries that belong to the chosen levels are probed in advance and used by the scheme at no additional cost.

The materialized candidate queries which EAGER-GS uses provides *two ways* to reduce the incurred cost: These two ways are:

- **Direct Hit:** The current candidate query R_i is already materialized, thus no need to call the database layer.
- **Aggregate Bounds:** Better aggregate bounds are found by utilizing the materialized queries which result in a better estimation of the deviation and ultimately lower cost.

In EAGER-S though, it only benefits from materialization if the current candidate query R_i is already materialized (i.e., direct hit). From experiments, we confirm and show that EAGER-GS benefits the most of the materialized queries when compared with EAGER-S. Also, we address the additional cost of materialization, and show through our experiments that materializing queries pay off after only a couple of runs.

3.4.3 Approximation Techniques for EAGER-GS

The design of EAGER-GS enables it to improve by means of approximation techniques. The proposed approximation techniques focus on reaching the objective with lower cost, while sometimes sacrificing on the deviation. In particular, we propose two approximation techniques for EAGER-GS, one to control the inverse relationship between minimizing the deviation and cost of refinement, while the other technique is to score cells in each level, and select only the most promising ones for next iterations.

The first technique relates to when EAGER-GS should stop the search process. As illustrated in Algorithm 3.2, EAGER-GS stops when it reaches the end level H, or in other words: the resolution of the lowest level in the pyramid. However, this condition was implemented so that we can have a comparable algorithm against the other baseline algorithms. Hence, we have improved EAGER-GS and changed its termination condition to be controlled by a parameter that is set by the user.

Algorithm 3.3 EAGER-GS with Stopping Condition (λ)

1: **Input:** Input Query I, G, α, δ, λ
2: **Output:** Refined Query R, Δ_{min}
3: G_I = getAggregate(I);
4: $\Delta_I^S = 0.0$; $\Delta_I = \alpha\Delta_I^S + (1-\alpha)\Delta_I^G$;
5: $\Delta_{min} = \Delta_I$; $\Delta_{TA} = 0.0$;
6: $H = \delta$; $h = 0.5$;
7: $SC = \Delta_I * \lambda$ ▷ Initializing Stopping Condition
8: $R_c \leftarrow$ GenerateQueries(I,h);
9: **while** ($\Delta_{min} \geq SC$) **do** ▷ Stopping Condition
10: ... ▷ Similar steps to SAQR-CS in Algorithm 3.2 lines 9-28
11: Return R_{opt}, Δ_{min};

The second approximation technique is inspired by another work [79] where only the top b cells with the highest scores are marked to be explored in the next iteration. Next, we explain in details the previously mentioned approximation techniques.

EAGER-GS Stopping Condition

As explained in Section 3.3.6, EAGER-GS stops when it hits the lowest possible resolution in the hierarchical structure of the search space. That design decision was made in order to have a fair and meaningful comparison between EAGER-GS and the other schemes.

However, as a standalone scheme, EAGER-GS's stopping condition is not related to the current resolution or the pyramid level. Therefore, we have introduced (λ) as a new parameter to control the stopping condition in EAGER-GS. Technically, (λ) provides a trade-off between the deviation and cost, i.e., it controls the inverse relationship between minimizing the deviation and cost of refinement.

Recall that EAGER-GS traverses the search space based on the pyramid structure, i.e., it starts from the highest level of the pyramid and goes down to the lowest level. Though, (λ) is not based on reaching a specific level in the pyramid. Instead, it is based on how much EAGER-GS has improved (or reduced) the input query's deviation, i.e., the scheme will stop the search once it finds a query with a deviation less than or equal to λ% of the input query's deviation. Specifically, EAGER-GS halts the search if the current refined query R_i has a deviation less than or equal to $(\Delta_I * \lambda)$. Setting (λ) to zero is an extreme case where the solution is not attainable. At the other extreme, setting (λ) to one means returning the same input query as the optimal solution.

In the experiments, we show the gains in efficiency against the loss in effectiveness controlled by λ.

Selecting Top Cells

In the cells exploration phase (Alg. 3.2 lines 26-28) EAGER-GS considers all cells in the current level to be explored. Nevertheless, selecting the most promising cells (according to an associated score) to be explored rather than selecting all the cells provides a tradeoff between efficiency and effectiveness.

Inspired by the work in [79], we adapted a similar logic, i.e., to score the cells and select only the $topb$ cells out of them for the next iteration. However, differently from that work, the score of each cell is based on our objective function presented in Eq. 3.2.5.

The score for a cell C is computed based on the two constraints: aggregation and similarity. Generally, we can estimate a *minimum bound* and a *maximum bound* deviation of those two constraints for any given C, similar to the query-level bounds described in Section 3.4.1, but at the cell-level.

The intuition is that a cell's score represents either the minimum deviation any query inside that cell could have (i.e., minimum bound deviation), or the maximum deviation any query inside that cell could have (i.e., maximum bound deviation). Specifically, minimum and maximum bound deviation of C are found by the following equation:

$$\Delta_C = \alpha\Delta_C^S + (1-\alpha)\Delta_C^G \tag{3.4.3}$$

On the one hand, Δ_C^{min} (i.e., minimum bound deviation) Δ_C^S and Δ_C^G are the minimum similarity deviation of C and the minimum aggregate deviation among all queries inside C, respectively. On the other hand, in case of Δ_C^{max}, Δ_C^S and Δ_C^G are the maximum similarity deviation of C and the maximum aggregate deviation among all queries inside C, respectively.

With those two bounds Δ_C^{min} and Δ_C^{max}, EAGER-GS has the capability to order cells depending upon Δ_C^{min}, Δ_C^{max}, or alternatively using the average of the them. Thus, we extend EAGER-GS to score the cells using the maximum and the average bounds, since these two provide better ordering of cells when compared to the minimum bound, as the minimum bound assumes a best case scenario for a cell, which might not be true. The steps for scoring the cells are listed in Algorithm 3.4. Notice how only the $topb$ cells are selected to generate the next candidate queries, as shown in lines 4-6.

To evaluate this approximation technique and show its benefits, we have implemented the TQGen scheme [79]. However, since this scheme was proposed to address the problem of cardinality-based query refinement only, its objective, pruning and scoring techniques were solely based on cardinality, without considering other aggregates or similarity at all.

Therefore, we have adjusted TQGen to address the aggregates and similarity constraints in its objective, and the pruning and scoring techniques, for the sake of a fair comparison to EAGER-GS.

Algorithm 3.4 Cell Scoring

```
1: for ( each cell C in h_prev ) do
2:     Δ_C ← Score(C);                          ▷ Score each cell using Eq. 3.4.3
3:     DS ← (C, Δ_C);                           ▷ Store in ordered data structure DS
4: for ( first topb cells C_1, C_2, ..., C_topb in DS ) do
5:     R_i ← C_i;                               ▷ Each cell represents a query
6:     R_c += GenerateQueries(R_i, h);          ▷ Generate queries for next level
```

Since TQGen is defined to work on multiple cardinality constraints for multiple sub-expressions queries, we mapped the multiple constraints to be the aggregate constraint G and similarity, i.e., two constraints. In the following, we explain the modifications that we made to TQGen to have a version comparable to EAGER-GS.

Firstly, the objective function used in TQGen is based on cardinality only. Therefore, we have replaced it with our objective function that considers similarity along with other aggregates, as specified in Eq. 3.2.5.

Secondly, TQGen utilizes a scoring function to score the cells in order to avoid an exhaustive search strategy. That is, during the exploration of the search space, all cells in level h are scored based on the number of cardinality constraints that a cell bounds, and then the $topb$ cells are selected for the next iteration. However, in our problem's setting, we have only one aggregate constraint for each input query. Given such setting, all cells that bound the constraint will have the exact same score. In such cases, TQGen uses a cardinality distance to score those cells given multiple constraints.

For the sake of a fair comparison, we have proposed similarity as a second constraint for scoring, along with the aggregate constraint. Hence, the score for a cell C in TQGen scheme becomes the weighted standard deviation of those two constraints. Specifically:

$$Score(C) = \frac{\sum_{i=1}^{k} w_i (x_i - \bar{x})^2}{\sum_{i=1}^{N} w_i} \tag{3.4.4}$$

Where k is the number of constraints, x_i and w_i are the value and weight of the constraint i, respectively.

Finally, we come to the third adjustment. In TQGen scheme, to prune a cell C, it firstly computes the error of the lower bound of C and then compares it against the best error found so far E_{best}. If it is worse than E_{best}, then the cell can be safely pruned and it will not be explored further. The error of the lower bound of C is similarly calculated to Δ_C^{min}. Thus, if Δ_C^{min} is higher than Δ_{min}, cell C is pruned and the candidate queries within that cell are not explored.

Parameter	Range	Default
Deviation weight (α)	0.0–1.0	0.5
Dimensions (d)	1–5	4
Number of input queries	-	100
Grid resolution (δ)	$1/2^5$–$1/2$	$1/2^3$
Predicates Weights (w_i)	0–1	$\frac{1}{d}$
Database Size ($\|B\|$)	100K, 1M, 4M, 8M	100K

Table 3.6: Evaluation Parameters

3.4.4 Experiments

We have implemented EAGER as a Java front-end on top of the MySQL database management system. We have evaluated the performance of EAGER under various workload settings. Table 3.6 summarizes all the controlling parameters used in our experiments.

Schemes: In our experiments, the following schemes are compared:

- **Hill Climbing (HC):** This is the scheme proposed in [14] to automatically generate queries with cardinality constraints for DBMS testing. However, in this work we have extended HC to use our similarity-aware objective function for different aggregates Eq. 3.2.5. HC navigates the search space depending upon an initial step in a greedy manner until no further reduction in deviation is attainable. Then, it reduces the step size and continue to greedily navigate the search space.

- **EAGER-S:** Our scheme, which utilizes similarity for navigating and pruning the search space.

- **EAGER-GS:** Our scheme, which extends EAGER-S and utilizes both similarity and aggregate bounds for navigating and pruning the search space.

- **TQGen:** A best-effort algorithm proposed in [79] which utilizes heuristics to find queries that approximately satisfy cardinality constraints. We discussed earlier our modified version of TQGen in Section 3.4.3.

To achieve a fair comparison between the different schemes, EAGER-GS is tuned so that the cell width at the bottom layer of the pyramid structure is equal to δ, while EAGER-S uses the cell width δ for its grid. Meanwhile, HC is modified to stop when its step size is equal to δ. Hence, the maximum resolution achieved by EAGER-GS and HC is the same as that of EAGER-S.

Databases: In our experiments, we use the publicly available database: Sloan Digital Sky Server (SDSS)[1]. Specifically, we are using the `Star` view from the `PhotoPrimary` table which has the brightness properties of stars along with their coordinates. Note that all the numerical columns in the databases are normalized in the range (0-1).

Queries: To cover a large spectrum of query contraction and expansion scenarios, we generated a set of 100 <query, aggregate> pairs. In particular, each pair is an input query together with its aggregate constraint. The queries are generated according to a uniform distribution over the query space, whereas the aggregate constraints are generated according to a uniform distribution over the database.

Performance Measures: We evaluate the performance of the above schemes in terms of the following metrics:

- Average Cost (C_R): That is the average number of probes (calls) made to the database *evaluation layer* for refining all the queries in the workload.
- Average deviation (Δ_R): That is the average deviation experienced by all the queries in the workload, where the deviation perceived by each query is computed according to Eq. 3.2.5.

Aggregate Operators: While we have experimented with all standard SQL aggregate operators: (`count, sum, min, max, avg`), we only report the results for (`count, max, avg`) as `sum` and `min` results are similar to `count` and `max`, respectively. If no aggregate operator is explicitly specified, it is `count` by default .

Impact of Similarity Weight (α)

In the first set of experiments, we measure the impact of the similarity weight (α) on our two performance measures (i.e., average deviation and cost) while $d = 2$ for three aggregate operators: `count, max, avg`.

The deviation Figures 3.15, 3.16 and 3.17 show a common trend for the average deviation: deviation increases while α approaches 0.5 - 0.6, then it starts to decrease. The reasons is, the two constraints (similarity and aggregate constraints) are at odds with each other, i.e., satisfying one of them conflicts with satisfying the other. The peak of this conflict is observed when $\alpha = 0.5 - 0.6$. Moreover, the figures shows how HC can easily get stuck at a local minima when $\alpha > 0$, which results in deviating from the optimal solution found by EAGER-S and EAGER-GS. For instance,

[1] http://www.sdss.org

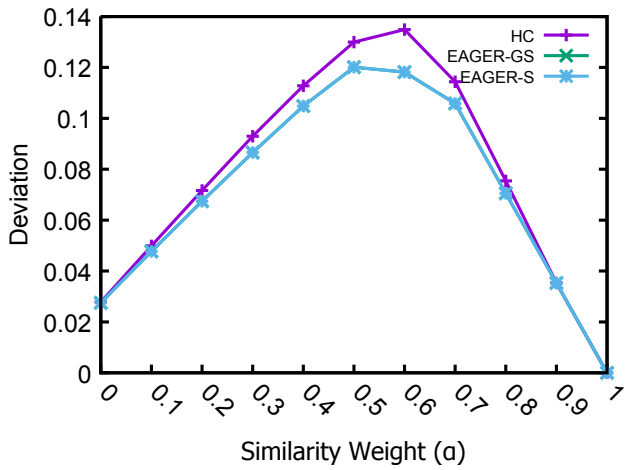

Figure 3.15: Average deviation while varying similarity weight α for `count`

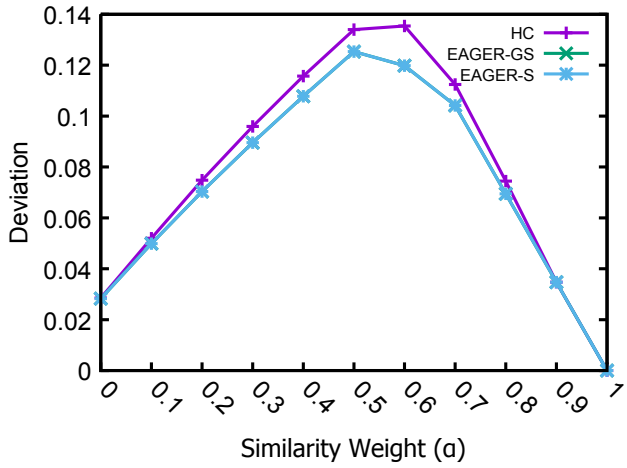

Figure 3.16: Average deviation while varying similarity weight α for `max`

when $\alpha = 0.5$, EAGER algorithms can improve the deviation by 14%-20% in some instants of the workload.

In regards to the second performance measure, i.e., average cost, Figures 3.18, 3.19 and 3.20 illustrate the efficiency of EAGER-GS when compared to EAGER-S and HC. This is due to the effective aggregate bounds and the hierarchical representation that EAGER-GS is based on. Also, the figures shows that the general cost trend of EAGER-S and EAGER-GS is different than that of HC. Specifically, the two algorithms EAGER-S and EAGER-GS benefit from higher similarity weight in pruning more candidate queries, while HC's pruning power seems relatively constant. This is because HC is implemented to stop when it reaches the same maximum resolution as in EAGER-S and EAGER-GS. Hence, even if HC get stuck at a local minima before reaching the maximum resolution, it will keep in exploring the search space with finer δs with the hope of finding a better solution.

As Figures 3.20 and 3.17 show, when $\alpha = 0$, EAGER-GS exhibits almost double the cost of HC to find the same solution. This is due to the loose aggregate bounds for `avg` defined in Eq. 3.4.1, which reflects how difficult it is to find tight bounds for `avg` aggregate constraint. Though, when $\alpha \geq 0.3$,

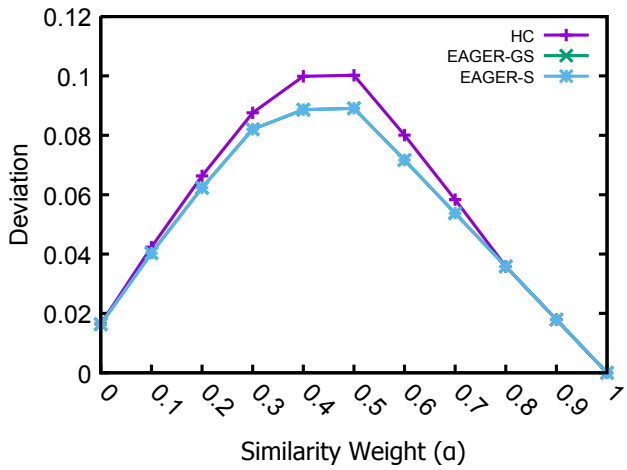

Figure 3.17: Average deviation while varying similarity weight α for `avg`

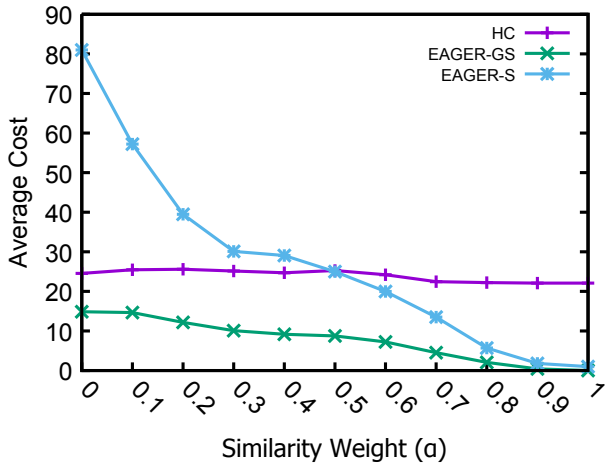

Figure 3.18: Average cost while varying similarity weight α for `count`

EAGER-GS dominates HC and EAGER-S in the two performance measures.

Impact of Dimensionality (d)

Next, we test the impact of dimensionality d on the performance of the compared algorithms while $\alpha = 0.5$. In Figures 3.21 and 3.22, HC manages to find the optimal solution when $d = 1$, just as EAGER algorithms, since it is not possible to get stuck at a local minima.

However, for $d > 1$, we can clearly see that HC deviates from the optimal solution found by EAGER algorithms. Note that since `avg` aggregate operator is not monotonic, even when there is one dimension, HC deviates from the optimal solution, as shown in Figure 3.23.

Figures 3.24, 3.25 and 3.26 are a numerical proof of the complexity for this refinement problem: the cost of navigating the search space increases exponentially with d. Also, Figures 3.24 and 3.25 show the dominating efficiency of EAGER-GS due to its effective pruning techniques. Though, the aggregate bounds for `avg` become less effective, specially with higher d. For instance, as Figure 3.26 shows, when $d = 5$, EAGER-GS's cost is almost the double of HC, due to the loose bounds defined in

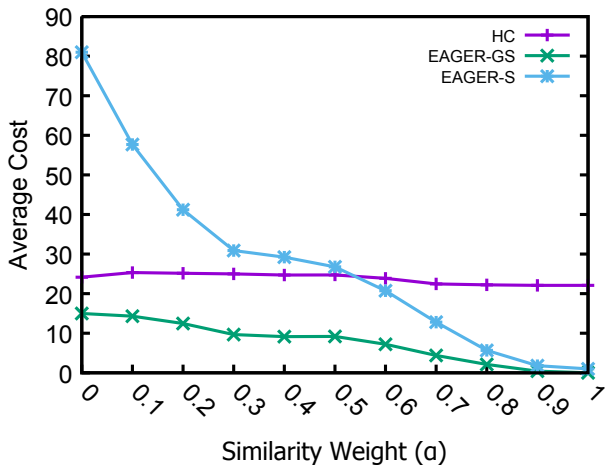

Figure 3.19: Average cost while varying similarity weight α for `max`

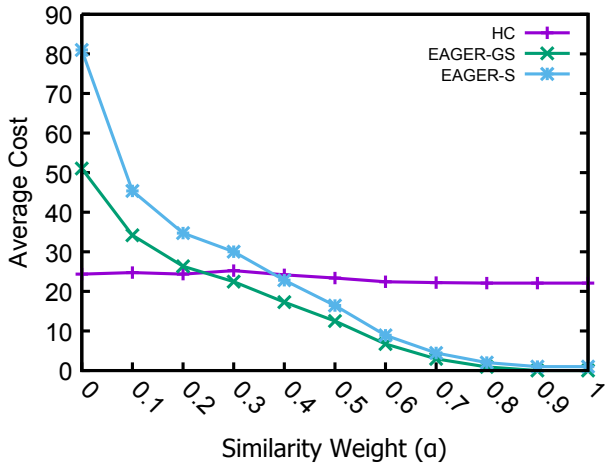

Figure 3.20: Average cost while varying similarity weight α for `avg`

Eq. 3.4.1 for `avg` aggregate operator. Nevertheless, despite the high cost, EAGER-GS achieves 24% better deviation than HC.

Impact of Grid Resolution (δ)

Recall that parameter δ specifies the grid resolution of the search space. As mentioned before, it was fixed to the default value throughout all the experiments introduced so far. In this experiment though, we want to examine the impact of δ on the performance metrics. Hence, we varied δ in the range $1/2^5 - 1/2$.

In Figure 3.28, the x-axis shows the variable δ and the average cost is shown on the y-axis. Clearly, the average cost drops significantly when δ is increased for all schemes. For instance, the average cost provided by EAGER-S is reduced by 70% when δ is increased from $\frac{1}{2^5}$ to $\frac{1}{2^4}$, while the reduction in the costs of HC and EAGER-GS is 10% and 53%, respectively.

The reason is, when δ is increased towards higher values, the total number of cells in the grid is decreased, leading to smaller number of cells to be scanned, thus, the average cost is tightly related

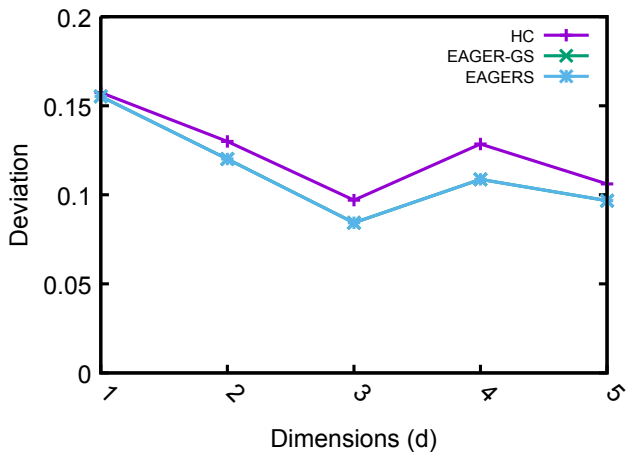

Figure 3.21: Average deviation while varying number of dimensions d for `count`

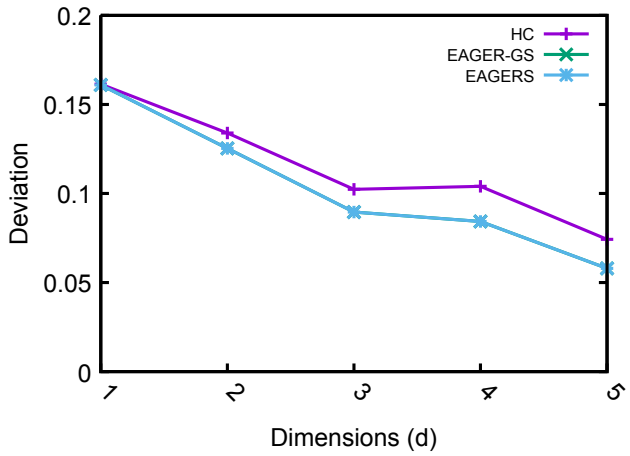

Figure 3.22: Average deviation while varying number of dimensions d for `max`

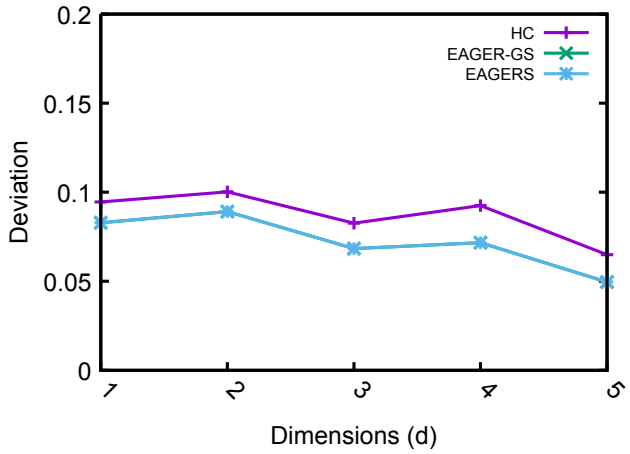

Figure 3.23: Average deviation while varying number of dimensions d for `avg`

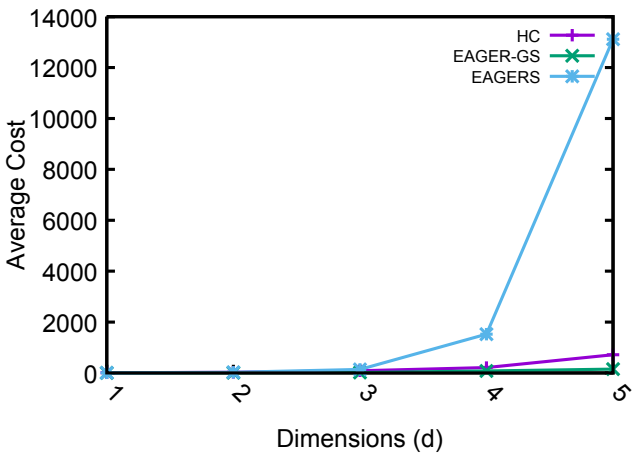

Figure 3.24: Average cost while varying number of dimensions d for `count`

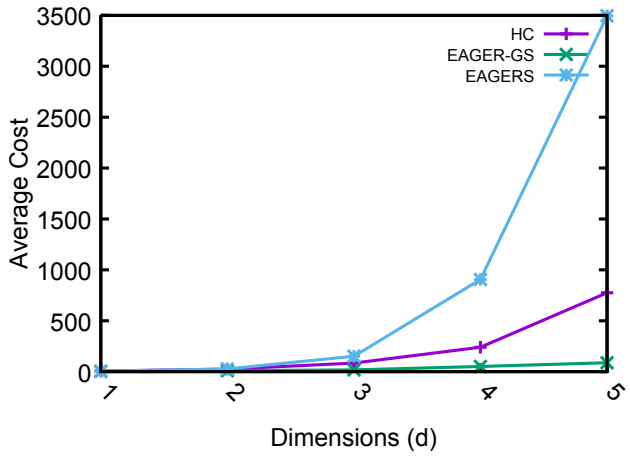

Figure 3.25: Average cost while varying number of dimensions d for `max`

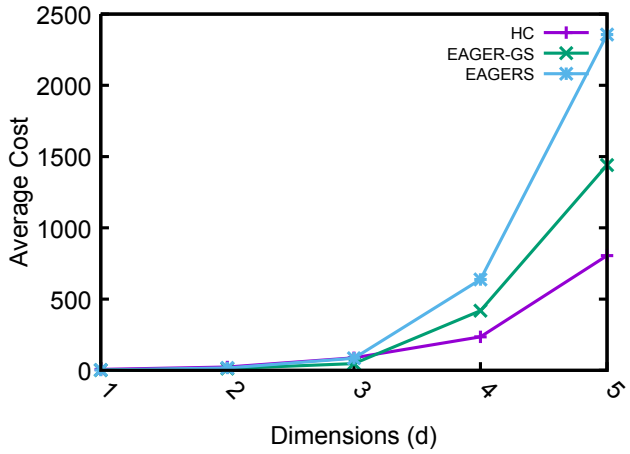

Figure 3.26: Average cost while varying number of dimensions d for `avg`

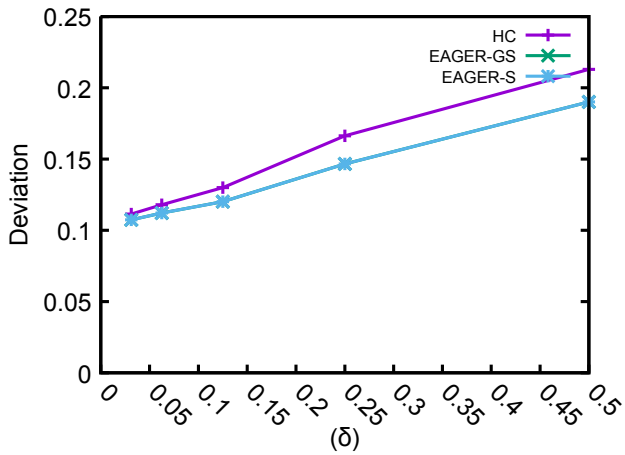

Figure 3.27: Average deviation while varying grid resolution δ

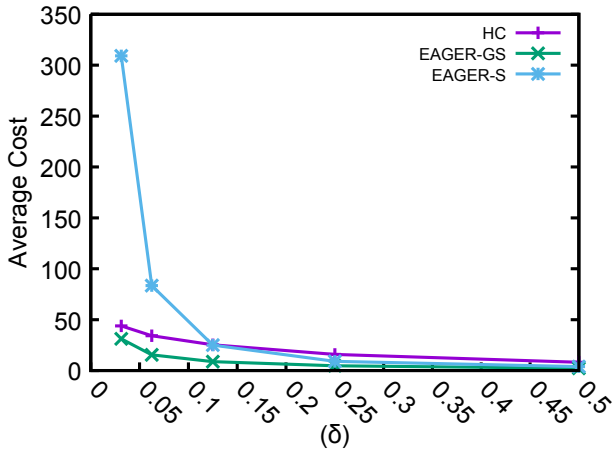

Figure 3.28: Average cost while varying grid resolution δ

to the search space resolution.

As shown in Figure 3.27, all three schemes exhibit direct correlation between δ and the average deviation. This relation is natural because when δ is increased, the search space is highly approximated, which essentially increases the probability of missing the exact target constraints, and vice versa. Looking at both Figures 3.28 and 3.27 uncovers the trade-off between the deviation and cost metrics which is controlled by δ.

Impact of Database Size

In this experiment, we have four versions of the SDSS database of sizes 100K, 1M, 4M and 8M tuples. As expected, Figure 3.29 shows the number of probes for each scheme is constant across all databases sizes. Clearly, this is because the search space of the refined queries remains the same for all sizes, i.e., size of $\mathbb{R}_\delta$ is independent of the database size. However, the database size determines the amount of data processed in each probe and in turn, the probing time (as shown in Table 3.7). In particular, for a machine loaded with Intel Core i7 3.40GHz CPU, 16.0 GB RAM, and Windows 7

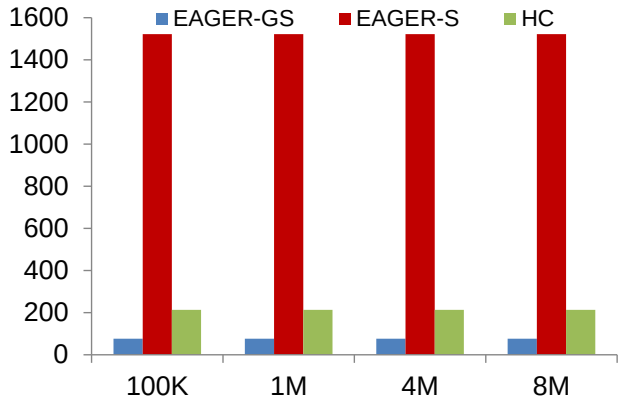

Figure 3.29: Average cost for SDSS database in different sizes

Dataset Size	100K	1M	4M	8M
Probe (ms)	33	419	2097	4207

Table 3.7: Time per probe (ms) for SDSS database in different sizes

OS, Table 3.7 shows the time per probe for the different databases sizes. Combining the results in Figure 3.29 and Table 3.7 shows that EAGER-GS allows for scalable and practical query refinement.

Reducing Cost by Materialization

This experiment shows how much cost reductions EAGER-GS can achieve from materialized candidate queries compared to EAGER-S and HC. The x-axis in Figures 3.30 and 3.31 show the number of materialized candidate queries, starting from the top level in the hierarchical representation and on to the bottom level. That is, the first point in the x-axis represent the number of materialized candidate queries that belong to the top level, while the last point in the x-axis represents the number of materialized candidate queries in all levels. Note that all compared algorithms use the same exact materialized queries in this experiment. The y-axis shows the average cost.

Clearly, from Figures 3.30 and 3.31, EAGER-GS demonstrates better utilization of materialized queries when compared with EAGER-S and HC. For example, when materializing only 6% of the overall candidate queries, EAGER-GS can achieve 67% reductions in cost, while EAGER-S and HC can only achieve linear cost reduction on the number of materialized queries. This shows how efficient EAGER-GS is when combined with materialization. Particularly, EAGER-GS takes advantage of the materialized queries in estimating the aggregated value of a given candidate query, which results in tighter aggregation bounds that lead to greater pruning power.

Obviously materializing queries in advance adds additional cost to refinement. However, when considering a scenario where more than 5 queries with constraints are submitted for refinement, the

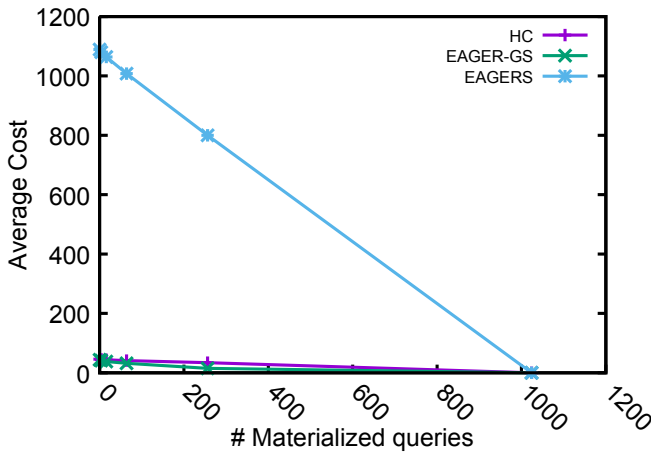

Figure 3.30: Average cost while varying number of materialized queries

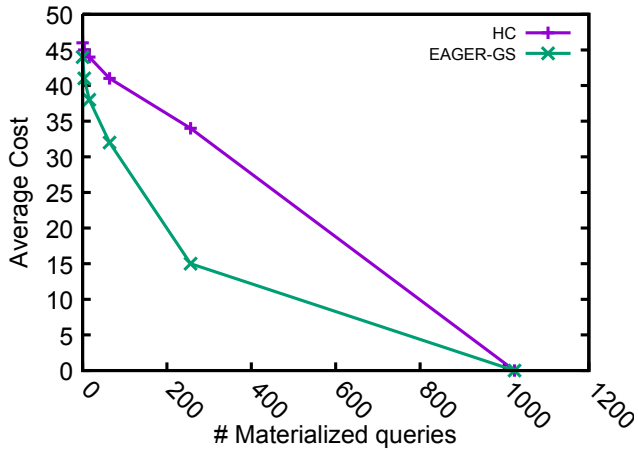

Figure 3.31: Average cost while varying number of materialized queries

average cost with materialization becomes less than when not materializing any queries in advance at all. This is shown in Figure 3.32. EAGER-GS with 6% materialized queries performs better than EAGER-GS without materialized queries, provided that more than 5 queries are submitted to the SDSS database.

Parameter (λ) as a Stopping Condition

The new stopping condition for EAGER-GS (i.e., λ) controls when the algorithm should stop searching for the optimal refined query. In the following, we investigate the behavior of this parameter and its effect on the cost and deviation of the scheme with the default settings and $\alpha = 0$.

Figure 3.34 shows that the deviation increases while increasing λ, whereas the cost decreases as Figure 3.33 shows. That is, the larger λ is, the earlier EAGER-GS stops traversing the search space, causing higher values of deviation. This is the classical behavior of any approximation parameter which controls the tradeoff between cost and accuracy (i.e., deviation). For example, when $\lambda = 0.05$, EAGER-GS finds an approximated solution with a loss of almost 86% on deviation, but 75% less cost

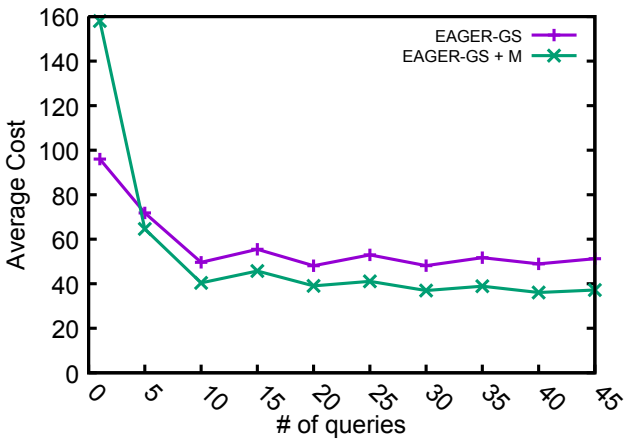

Figure 3.32: Average cost while varying number of submitted queries

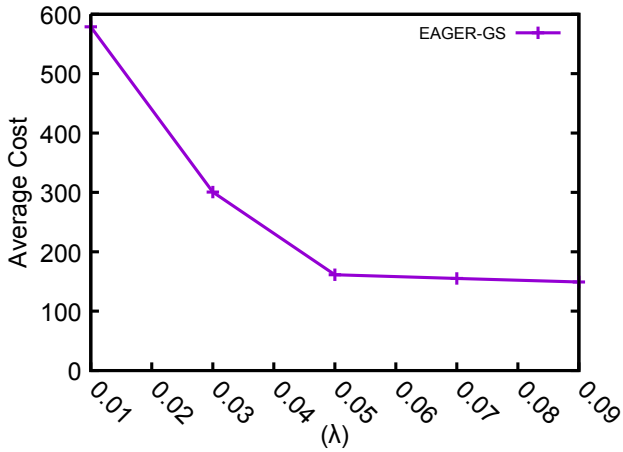

Figure 3.33: Average cost while varying threshold λ

when compared to $\lambda = 0.01$.

As a guideline for setting λ, users should set higher values of λ to get prompt and approximated results. Conversely, setting lower values to λ provides more accurate approximated result at the expense of cost.

EAGER-GS with Cells Scoring

Instead of exploring all cells in the search space, EAGER-GS in this experiment scores the cells using the minimum or average bounds, then it chooses only the *topb* cells for further exploration. To show EAGER-GS efficiency gains when using scoring, we implemented TQGen [79] and modified it to compare those two schemes, as explained earlier in Section 3.4.3. Note that TQGen uses a grid of uniform cells to represent the search space, and defines a parameter called *segments* k to partition a cell into k^d cells. Having $k = 2$ is similar to how EAGER-GS represents the search space (i.e., pyramid structure). For this experiment, we set $d = 2, \alpha = 0.5$ and $\delta = \frac{1}{2^5}$.

Figures 3.35, 3.36 and 3.37 show the average deviation while varying *topb* for TQGen and three

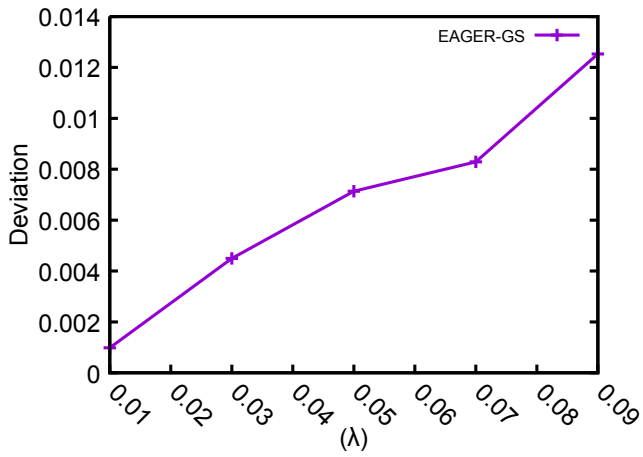

Figure 3.34: Average deviation while varying threshold λ

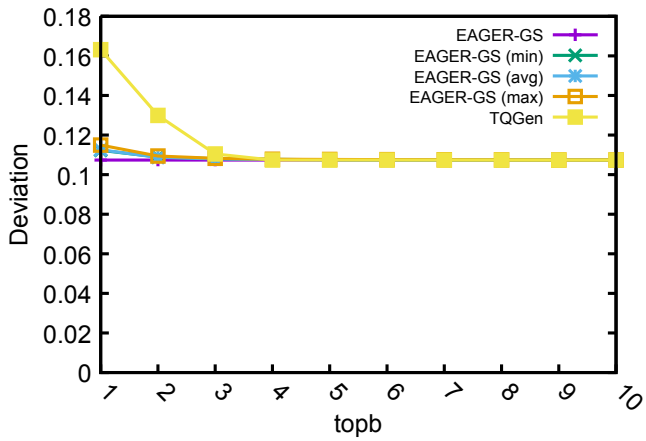

Figure 3.35: Average deviation while varying *topb* for `count`

versions of EAGER-GS that score cells based on minimum, maximum and average cell deviation by Eq. 3.4.3. Also, the figures show EAGER-GS without scoring as a benchmark for comparison.

Interestingly, when $topb \geq 4$, TQGen finds the same exact optimal solution found by all the different versions of EAGER-GS, although at much higher cost, as shown by Figures 3.38, 3.39 and 3.40. Essentially, when $topb \geq 4$, TQGen will not benefit from its scoring approach since it will select all 2^2 cells (generated from partitioning a cell) regardless of their scores, as long as they are not pruned. On the other hand, EAGER-GS with its three different scoring versions, utilizes cells scores by selecting *topb* cells out of $(2^H)^2$ cells in a level H, resulting in a much better approximation when compared with TQGen. For example, if all algorithms to select only the top scored cell, EAGER-GS (with different scoring versions) reduces cost by almost 66%, while deviating from the optimal solution by only 7%. However, TQGen deviates almost 34% from the optimal solution, with a cost reduction of 22%, when compared with EAGER-GS.

As Figures 3.37 and 3.40 shows, for aggregate operator `avg`, EAGER-GS with its different scoring versions is able to find almost the same optimal solution, with up to 89% lower cost than EAGER-GS.

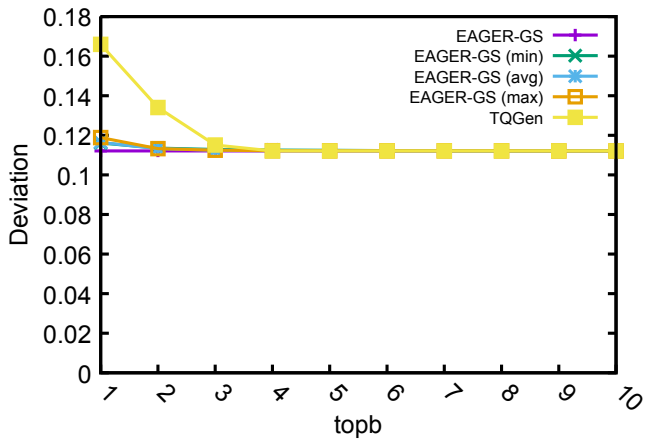

Figure 3.36: Average deviation while varying *topb* for `max`

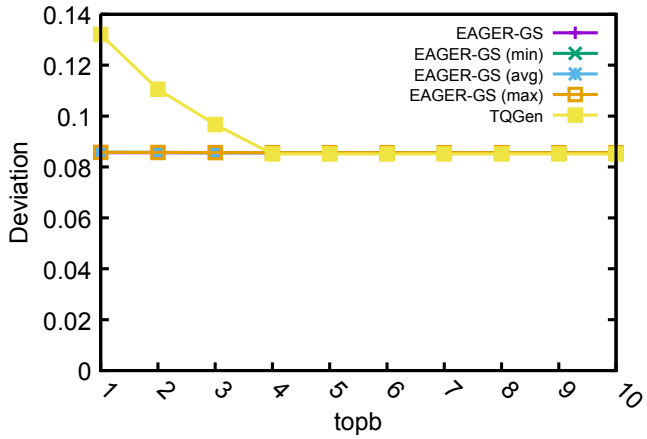

Figure 3.37: Average deviation while varying *topb* for `avg`

The reason behind this large cost reduction is that only *topb* cells are chosen for exploration in each iteration, which means a small number of candidate queries are probed, hence, overcoming the limitation of the loose `avg` bounds without any significant reduction in deviation.

Top-K Refined Queries

So far in this work, we assume that users are looking for only one refined query that has the optimal deviation. Nevertheless, in other cases, users are looking for K choices of refined queries which minimize the objective at the different levels. Naturally, EAGER schemes support finding those K choices. Hence, we conducted an experiment to see how sensitive our schemes are to K. That is, we want to know how does K effect the performance of EAGER schemes. Using the default experiment settings and workload, we measured the pruning power of EAGER-GS and EAGER-S while varying K from 1 to 10.

As Figure 3.41 shows, EAGER-GS outperforms EAGER-S in pruning most of the search space while attaining the same top K refined queries. For example, when K equals 10, EAGER-GS

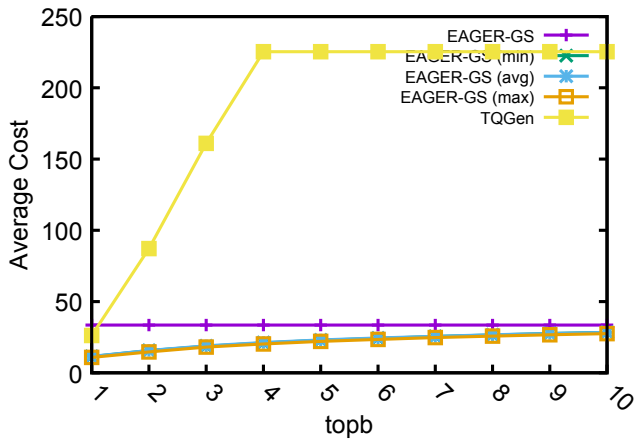

Figure 3.38: Average cost while varying *topb* for `count`

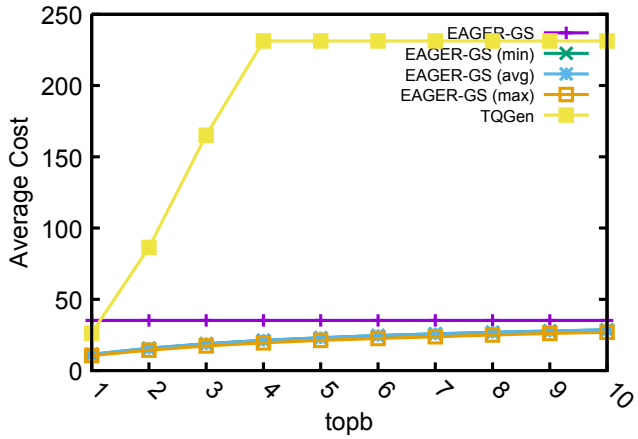

Figure 3.39: Average cost while varying *topb* for `max`

achieves 19% better pruning power than EAGER-S. This emphasizes the benefits of the used pruning techniques in EAGER-GS, namely: the hierarchical structure of the search space and the cardinality based pruning. Though, for both EAGER-GS and EAGER-S, the pruning power decreases with increasing K, because in order to find more K refined queries, both schemes need to explore more candidate queries.

Results Discussion

As illustrated above, EAGER-GS dominates EAGER-S and HC for different parameters settings and for different aggregate constraints. However, in case of the `avg` aggregate constraint, HC outperforms EAGER-GS in terms of cost since the aggregate bounds for `avg` are very loose, i.e., EAGER-GS exhibits lower pruning power under this particular aggregate constraint. EAGER-GS benefits the most from candidate queries materialization while HC and EAGER-S only achieve linear cost reduction. The stopping condition λ offers a tuning parameter for EAGER-GS to trade off cost for deviation. The approximated versions of EAGER-GS provide efficient solutions to the `avg` aggregate constraint

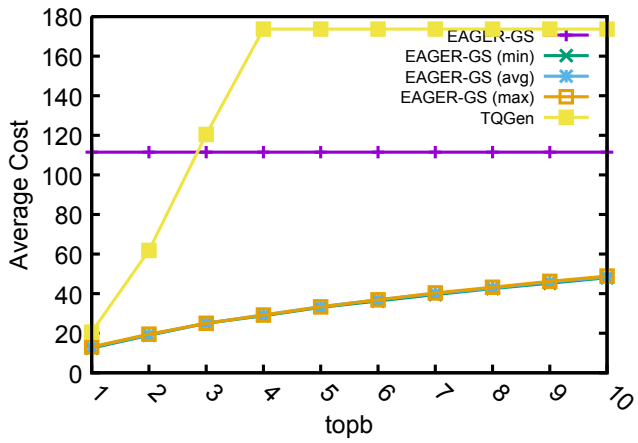

Figure 3.40: Average cost while varying *topb* for avg

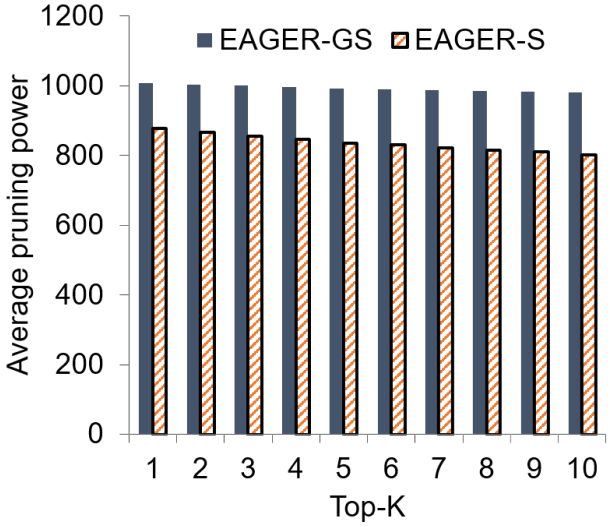

Figure 3.41: Average pruning power with different Top-*K*

and also outperforms TQGen when setting *topb* to a small number, i.e, EAGER-GS is set to explore only a small number of cells in each level in the grid.

3.5 Objective-aware Range Query Refinement

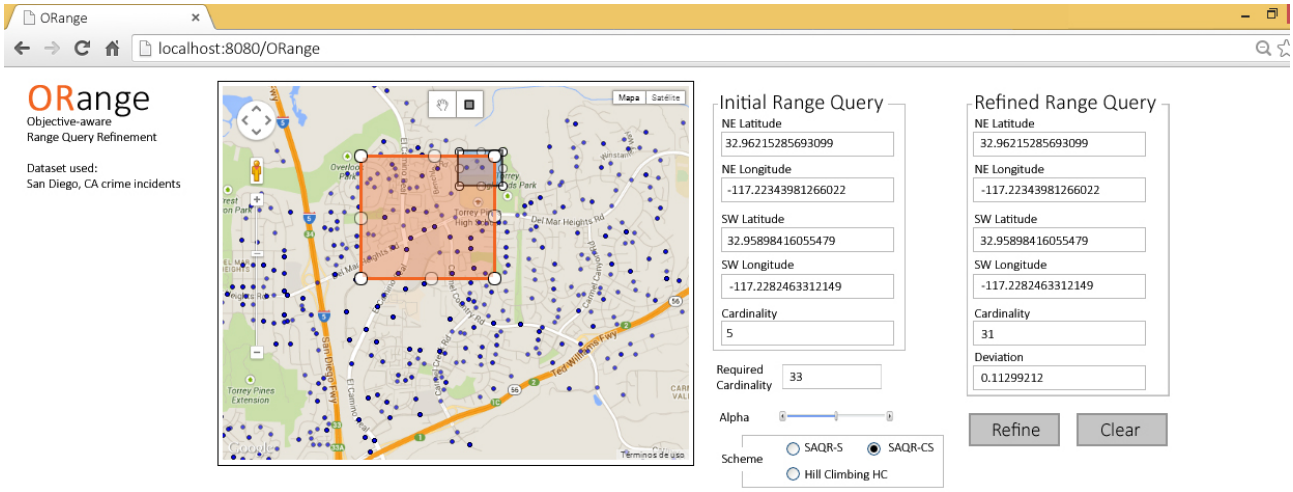

Figure 3.42: ORange's web interface

ORange (**O**bjective-aware **Range** Query Refinement) is a web-based tool for range queries refinement. Essentially, ORange refines a range query to meet a specified cardinality constraint while taking into account the (dis)similarity between the initial query and its corresponding refined version.

To showcase SAQR's efficiency and benefits, we designed and developed: *ORange*: an application that guides police coordinators in allocating police officers into service zones, such that each police officer has a specific capacity (number of incidents) [121, 34]. That is, each police officer can only handle K incidents at a given time, therefore, when allocating a service zone to her, it must contain K incidents. More or less incidents in her service zone corresponds to a drop of quality of service or waste of resources, respectively. Accordingly, we employed the proposed SAQR schemes in Section 3.3 to refine service zones given their capacity (cardinality) constraints. We used the historical dataset of crime incidents of the city of San Diego, CA in USA [2] to estimate the number of incidents, therefore, this application is based on that city. Nonetheless, users can upload their own datasets to utilize ORange capabilities.

The capabilities of ORange can be summarized as follows:

- A police coordinator can enter a cardinality constraint and visually selects an initial range query (service zone) on a real map for a police officer.
- A police coordinator can see the new refined query provided by the application on the same input map, such that the new query satisfies the cardinality constraint.

[2] Extracted from clarinova.com-crime-incidents-casnd-7ba4. San Diego Regional Data Library. 2013-08-07 http://sandiegodata.org

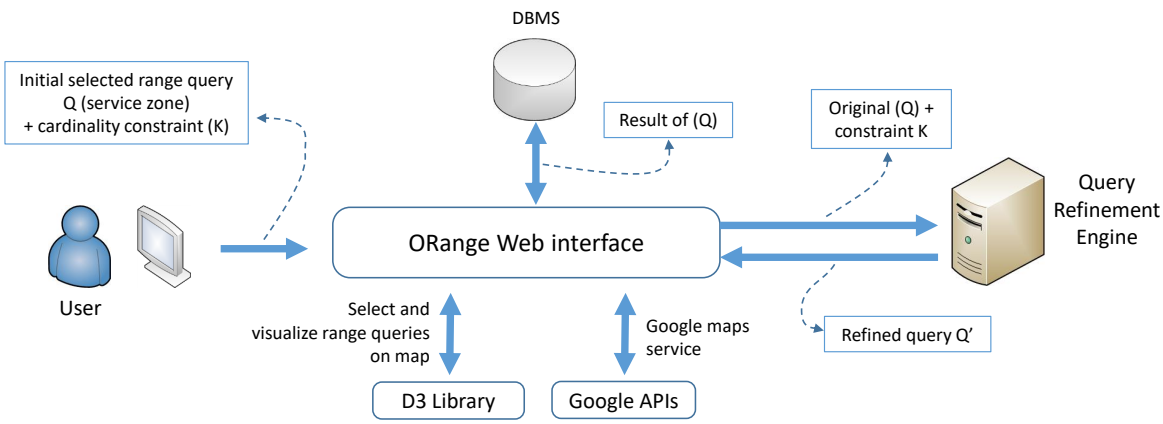

Figure 3.43: ORange's complete system architecture

- Performance details (cost and deviation) are shown for each scheme to judge and compare them against a baseline heuristic algorithm: Hill Climbing.

In the next section, we firstly introduce the application's architecture, then we briefly present the application's setup and the used dataset. As for the underlying schemes of ORange: SAQR-S and SAQR-CS, which leverage and exploit the distance and cardinality constraints to effectively prune the search space, we refer the reader to Section 3.3 for more detailed description of those schemes.

3.5.1 ORange Architecture

Figure 3.43 shows a detailed architecture of ORange and its building modules. *ORange* communicates with the user through a web interface and receives the input as: a selected range query (service zone) and a constraint K. The former is captured by two modules: Google Maps APIs and D3 Library, to show a real world map and to draw a rectangular area, respectively. Then those input data are fed to the *Query Refinement Engine*. When the refinement engine finds the refined query, it sends it back to the web interface to display (with the help of Google Maps APIs and D3 Library) the new refined service zone for the user to see. If the user is not satisfied with the result, she can issue a new query and provide new constraints.

3.5.2 Application Setup

The ORange application is built as a client-server application with a front-end that handles all presentation tasks and a back-end for processing data. The front-end is a web interface which consists of an HTML page that provides the capabilities of communicating input data from users to the system and showing output to users in a suitable way (See Figure 3.42). A visualization library called *D3 Library* is used to aid in drawing range queries as rectangular shapes on a real world map provided

Attribute	Description
date	ISO date, in YY-MM-DD format
year	Four digit year
month	Month number extracted from the date
day	Day number, starting from Jan 1, 2000
dow	Day of week, as a number. 0 is Sunday
time	Time, in H:MM:SS format
type	Crime category, provided by SANDAG
address	Block address, street and city name
Latitude	Provided by the geocoder
Longitude	Provided by the geocoder
desc	Long description of incident

Table 3.8: Schema of used dataset *SD_incidents_100k*.

by *Google Maps APIs*. The controlling parameters: cardinality constraint K, α and selected scheme are collected from users through HTML input tags. While the previously mentioned presentation tasks are all located in the front-end, all processing tasks are located in the back-end. Specifically, the refinement engine is located in the back-end and is implemented using Java. Its job is to receive the input parameters (query and control parameters) and return the optimal refined query and the performance indicators to the front-end for presentation. Specifically, the front-end will show the quality of refinement and the cost metric (via charts in a dashboard) for all schemes to users to comapre. All data is stored in MySQL DBMS, and as explained above, we are using a historical dataset of crime incidents of the city of San Diego, CA in USA. That dataset consists of one relation of 100k incidents. Each incident is represented by multiple attributes, however, we are only concerned with the longitude and latitude attributes of the crime incident. A partial schema of the relation is shown in Table 3.8.

3.5.3 Step-by-step Example

As shown earlier, the dataset used represents the locations of historical crime incidents within the city of San Diego. Users can allocate a service zone for a police officer, i.e., that police officer will be

in charge of any incident reported in his service zone and he must respond to it. Provided with the application interface in Figure 3.42, the user should perform the following four steps:

Step 1: The user will use a selection tool to select a rectangular area in the map of San Diego which represents the desired service zone. To keep the user more informed, the NE (North east) and SW (South west) coordinates of the selected zone are also shown on the text boxes.

Step 2: Once the zone is selected, the user then enters the cardinality constraint and selects the controlling parameters (α and refinement scheme).

Step 3: To execute the refinement process, the user clicks on the Refine button, and waits until the processing is finished. This is when the refinement module takes over and starts navigating the search space looking for a refined query that has the minimum overall deviation.

Step 4: As soon as the refinement process finishes, the new returned area is drawn on the same input map, but with different color to distinction between the initial and refined service zones. Also, the deviation from the initial selected area is shown in the text box as a normalized value between [0-1].

3.6 Summary

In this chapter, we presented our contributions towards the Similarity-aware, Aggregate-based Query Refinement problem in which users specify aggregate constraints for their queries. Additionally, we proposed to include the similarity of the refined query to the input query as an objective in refinement, aiming to increase users' satisfaction with the refined queries. As a first step, we proposed a declarative query model to specify all relevant parameters for refining a query in an additional SQL-alike clause. Given a special case of Aggregate-based Query Refinement problem, we initially focused on the cardinality constraint and presented SAQR as an efficient scheme for cardinality-based query refinement. By utilizing similarity and cardinality based pruning techniques to minimize the incurred costs, SAQR scheme efficiently returns a refined query that balances the tradeoff between satisfying the imposed cardinality and similarity constraints to maximize the overall benefit to the user.

Then, we extended the special case of the cardinality constraint by including the SQL standard aggregate functions `sum`, `avg`, `min`, `max` as constrains for refinement, which is the general case in the Similarity-aware, Aggregate-based Query Refinement problem. For that general case, we presented EAGER schemes that extend on SAQR schemes to satisfy the aggregate constraints. Additionally, EAGER schemes implement unique optimization and approximation techniques to minimize the costs incurred in exploring the search space. These techniques include strategic materialization of candidate queries, scoring candidate queries based on similarity and aggregate

bounds, and controlling the stopping condition of EAGER which provides a trade-off between the deviation and cost.

Finally, we showcased SAQR in a web-based application called ORange. ORange's goal is to aid police coordinators in allocating service zones to police officers given a capacity (i.e., cardinality) constraint. To achieve this goal, ORange presents the user with a real map so that an initial service zone can be selected for an officer. Subsequently, ORange employs SAQR schemes to produce a refined service zone that satisfies the required cardinality and similarity constraints on the initial selected service zone.

CHAPTER 4

Similarity-aware Correlation-based Query Refinement

4.1 Overview

Exploration of time series data [114, 122, 97, 135] is evident in many domains (e.g., financial, medical and environmental domains). One of the key analysis tasks in these domains is to detect patterns or anomalies among multiple time series [76, 72]. For instance, in the financial domain investors intensively analyze the stocks market daily close prices to understand the patterns of the stocks market and to make investment-related decisions based on some discovered patterns [89, 39]. Moreover, time series data in the medical domain contain vital information. Examining and analyzing these series to detect unusual patterns can significantly save a human life [122, 89].

In this chapter, we present our novel contribution for the Correlation-based Query Refinement problem in the context of sequential data (i.e., time series data). In particular, we present the Similarity-aware, Correlation-based Query Refinement problem (SCQR) which aims to efficiently refine a user's query to satisfy a certain targeted pattern while maximizing the similarity between the refined query and the initial one. In this problem, a pattern is defined as the pairwise correlation of all time series pairs (i.e., a correlation matrix M), where correlation of a pair is computed using the well-known Pearson's correlation coefficient [101, 36, 99, 39, 62, 1, 38]. The matrix M_{Q_1} in Figure 1.5 is one example of a correlation matrix for three time series.

The organization of this chapter is as follows: Section 4.2 presents preliminaries and formally defines the Similarity-aware, Correlation-based Query Refinement problem, in which users' queries are refined to satisfy pairwise correlation constraints. Then, Section 4.3 proposes the RELATE scheme based on the classical tree traversal methods BFS and DFS to be as solutions for this problem. Afterward, Section 4.3.5 discusses in detail our proposed optimization techniques for

Symbol	Description
$R = \{T_1, T_2, ..., T_n\}$	Relation R contains n time series
m	Length of time series in R
n	Number of time series in R
$\rho(T_i, T_j)$	Pairwise correlation of T_i, T_j
M_t	Target correlation matrix of size $n \times n$
$Q[s,e]$	Range selection query on `timestamp` domain with sub-interval $[s,e]$
Q_I	User's input query
M_Q	Correlation matrix of Q
M_t	Target correlation matrix
$C(Q', M_t)$	Tightness of $M_{Q'}$ to M_t
$S(Q', Q)$	Similarity of Q' to Q
λ	Similarity weight
Q^*	Optimal refined query

Table 4.1: Summary of Symbols

RELATE scheme. The proposed two-level pruning techniques enable RELATE to avoid processing unqualified queries and to early abandon the correlation computations for some pairs of time series using a monotonic property. Finally, Section 4.4 presents the results of extensive experiments to show the efficiency of our proposed schemes on synthetic and real datasets, and compare them to state-of-the-art algorithm.

4.2 Preliminaries

The task of detecting patterns and anomalies within time series data is fairly common in many domains [114, 122, 97, 21, 130] such as the financial, medical, environmental and network domains. Previous works have shown the importance of computing pairwise correlation to detect patterns in time series data for various applications [70, 87, 39, 100]. For instance, computing pairwise correlation for all time series pairs using the whole time interval [87] (or a fixed sub-interval [39, 100]) is extremely beneficial in data centre monitoring systems to discover correlated servers.

Computing pairwise correlation for *all possible sub-intervals* to discover patterns is a computationally challenging problem. Works such as [70, 71] aim to find the longest correlated subsequence (i.e., sub-interval) of two time series, while [89] aims to find the maximum sub-interval

with the highest pairwise correlation for a given pair. It should be clear that exploration of time series based on the whole time interval (rather than sub-intervals) is vulnerable to the classical Yule-Simpson effect [8, 40]. At the same time, exploration of all possible time sub-intervals is a much harder problem, since the number of sub-intervals increases quadratically with the length of time series. The toy example we introduced in Section 1.3.2 shows the usefulness and the challenges of exploring time series based on sub-intervals.

Accordingly, we propose the Similarity-aware, Correlation-based Query Refinement problem, in which a user's query is refined to satisfy user-defined pairwise correlation constraints. As we see next, achieving the goal of this problem requires computing the pairwise correlation values of *all time series pairs* for *all possible time sub-intervals*. The applications of this problem are fairly prevalent in data centre management systems [73] where users analyze servers' loads collectively to discover patterns and anomalies based on the pairwise correlation values [87, 102]. We present the details of this problem in the following section. All used symbols are listed in Table 4.1 with their descriptions.

4.2.1 Problem Definition

In this section we formally define the Similarity-aware, Correlation-based Query Refinement problem. As in [74, 27], we assume the presence of n synchronized, equal length time series stored in a flat relational table $R : \{T_1, T_2, ..., T_n\}$ where each $T_i \in R$ contains m real values $\{v_1^i, v_2^i, ..., v_m^i\}$ such that $v_j^i \in \mathbb{R}$ is the jth value with time stamp j in T_i.

Users explore R by submitting SQL range queries on the `timestamp` attribute to select a time sub-interval $[s, e]$ of the time series in R, then further analyze the results based on the pairwise correlation of all time series in R, i.e., correlation matrix M. Next, we define the basic notions of this problem, then later give the formal problem definition.

Definition 4.1. *Q is a range selection query on the `timestamp` attribute:*

$$\sigma_{s \leq timestamp \leq e}(R)$$

such that $1 \leq s \leq e \leq m$.

For ease of readability, a query will be denoted as $Q[s, e]$, or Q if the time sub-interval $[s, e]$ is of no importance.

As we see next, achieving the goal of the Similarity-aware, Correlation-based Query Refinement problem implies iterating over all possible sub-intervals within [1-m]. Hence, the number of possible refined queries is $\frac{m(m-1)}{2}$, which increases quadratically with the length of time series m. This

observation renders the problem at hand to be computationally hard because the correlation matrix M will be computed for each one of these queries.

Definition 4.2. *Correlation matrix M is a symmetric matrix of size $n \times n$. Each entry $M[i][j] \in M$ is precisely the pairwise correlation of T_i and T_j.*

It shall be clear from Definition 4.2 that $M[i][j] = M[j][i], \forall i = 1,2,...,n-1; \forall j = i+1, i+2, ..., n$.

Definition 4.3. *The pairwise correlation of T_i and T_j of length ℓ is measured by the Pearson's correlation coefficient $\rho(T_i, T_j)$.*

$$\rho(T_i, T_j) = \frac{\ell \sum_{k=1}^{\ell} v_k^i v_k^j - \sum_{k=1}^{\ell} v_k^i \sum_{k=1}^{\ell} v_k^j}{\sqrt{\ell \sum_{k=1}^{\ell} (v_k^i)^2 - (\sum_{k=1}^{\ell} v_k^i)^2} \sqrt{\ell \sum_{k=1}^{\ell} (v_k^j)^2 - (\sum_{k=1}^{\ell} v_k^j)^2}} \tag{4.2.1}$$

Note that $M[i][i] = 1, \forall i = 1,2,...,n$. We are now in place to formally define the problem at hand:

Definition 4.4. ***Similarity-aware, Correlation-based Query Refinement Problem**: Given an input query Q_I and a target correlation matrix M_t. The goal is to automatically refine Q_I to Q^* such that $f(Q^*)$ is maximized.*

$$f(Q^*) = f(Q_I, Q^*, M_t) = \lambda S(Q_I, Q^*) + (1-\lambda) C(Q^*, M_t) \tag{4.2.2}$$

$$S(Q_I, Q^*) = 1 - \frac{1}{1 + e^{-d(Q_I, Q^*)}} \tag{4.2.3}$$

$$C(Q^*, M_t) = 1 - (\frac{1}{z} \sum_{i=0}^{n-1} \sum_{j=i+1}^{n} (M_t[i][j] - M_{Q^*}[i][j])^2) \tag{4.2.4}$$

where z is a normalization factor, and:

$$d(Q_I, Q^*) = |Q_I.s - Q^*.s| + |Q_I.e - Q^*.e| \tag{4.2.5}$$

As stated in Definition 4.4, the optimal solution Q^* is the one with the maximum similarity $S()$ and the maximum closeness to the target correlation matrix $C()$, balanced by a user parameter λ. Ensuring maximum similarity of Q^* to Q_I is useful when users are interested for a time interval that is close from Q_I's interval. Similarly, ensuring maximum closeness to M_t is important to maximally achieve the target.

Modeling the similarity of a query to Q_I (i.e., $S()$) as a Sigmoid function [134] on the `timestamp` attribute has two advantages: it is a parameter-free function, and it expresses users interests to the

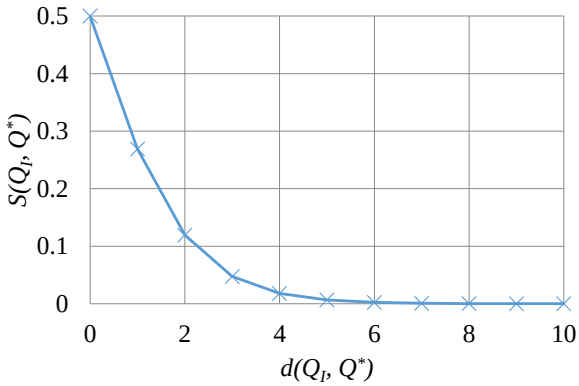

Figure 4.1: The similarity $S(Q_I, Q^*)$ decreases very quickly when distance $d(Q_I, Q^*)$ increases

input query. A query's sub-interval that is close from the user's input query's sub-interval (i.e., a small distance $d()$) should have more benefit than another query that is far from Q_I (i.e., a large distance). Figure 4.1 shows a visual intuition of the Sigmoid function: as the distance between Q_I and Q^* increases, the similarity decreases very quickly.

As for $C()$, we use the Sum of Square Errors (SSE) since it indicates the tightness of M_t to a matrix M_Q of a candidate query Q. Its normalized value ranges between [0-1], where a small value denotes a tight fit of M_Q to the target M_t.

In Definition 4.4, refining Q_I implies modifying Q_I's time interval $[s,e]$. We concretely define next the possible modification operations.

4.2.2 Refining a Sub-Interval

To refine a query Q with a selection predicate on the `timestamp` attribute, one of these two operations are applied on that time interval $[s,e]$, as shown in Figure 4.2:

1. **Expansion:** to expand $[s,e]$ from either sides s or e by δ. For instance, $[\hat{s},e]$ is expanded from s side by δ such that $\hat{s} = s - \delta$ while $[s,\hat{e}]$ is expanded from e side by δ such that $\hat{e} = e + \delta$. We encode those two operations as *LE* (left expansion) and *RE* (right expansion).

2. **Contraction:** to contract $[s,e]$ from either sides s or e by δ. For instance, $[\hat{s},e]$ is contracted from s side by δ such that $\hat{s} = s + \delta$ while $[s,\hat{e}]$ is contracted from e side by δ such that $\hat{e} = e - \delta$. Similarly, we encode those two operations as *LC* (left contraction) and *RC* (right contraction).

To ensure no possible candidate queries are missed, we set $\delta = 1$. Setting $\delta > 1$ has the effect to reduce the number of candidate queries. While it is tempting to reduce the number of candidate

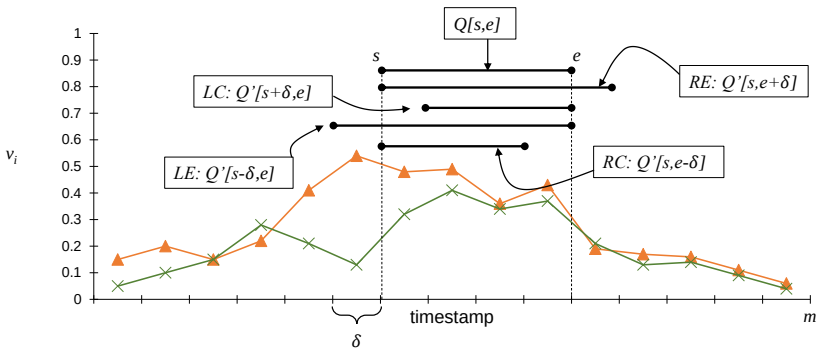

Figure 4.2: Refining a query $Q[s,e]$ implies refining its time sub-interval. Four candidate queries are generated by applying LC, LE, RC and RE on Q_I's sub-interval

queries (for efficiency reasons), this will introduce approximate solutions since an optimal query might be one of those candidates that were skipped.

Achieving the goal of the problem defined in 4.4 requires recursively applying the refinement operations LE, RE, LC, RC on Q_I to generated all possible sub-intervals and computing their objective function Eq. 4.2.2. However, this exhaustive task of finding a query with a specific correlation matrix is computationally challenging for the following reasons:

1. An algorithm has to go through all possible candidate queries (sub-intervals), which increase quadratically with the length of time series,

2. The number of pairs in the correlation matrix increases quadratically as well with the number of time series,

3. Computing correlation from scratch hinder the exploration process, while caching some results to boost correlation computations is limited by the amount of available memory.

To emphasize the computational costs of a correlation matrix, we show in Figure 4.3 the CPU and I/O time (normalized) for computing a single matrix in a traditional PC. As shown in the figure, the time it takes to compute a correlation matrix (CPU time) exceeds the I/O time when there are thousands of time series. For instance, when there are $n \geq 1000$ time series, CPU time dominates I/O time. [131, 7] also notes the daunting complexity of computing the pairwise correlation of all time series pairs in another settings were the z-normalized Euclidean distance is used in place of the Pearson's coefficient.

Towards finding the optimal solution Q^*, we propose an efficient scheme with innovative optimization techniques. Our propose scheme, RELATE, adopts the classical tree traversal strategies: Breadth First (BFS) and Depth-First (DFS), which allow for innovative optimization techniques to be incorporated in to efficiently find Q^* without a compromise on the solution accuracy. RELATE takes

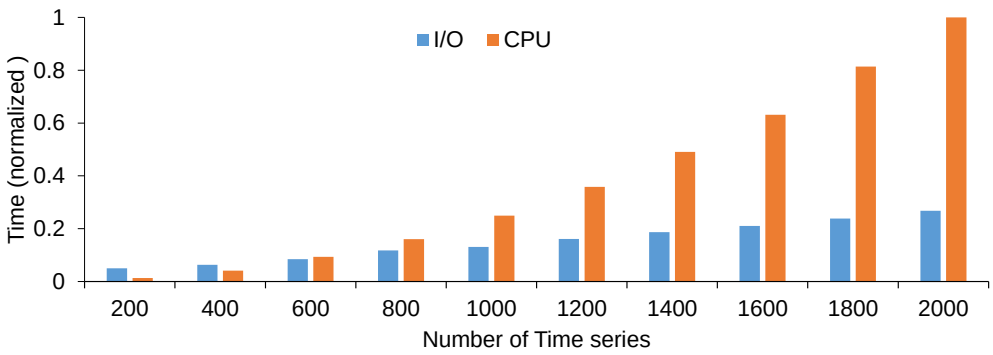

Figure 4.3: Computational time (CPU) to compute M dominates I/O time when there is a large number of series $n \geq 1000$

advantage of a monotonic propriety to avoid processing some of the candidate queries by applying two-level pruning techniques, and to cache some of the computed results which might be helpful for incrementally computing correlation of later queries. Details are in the following sections.

4.3 RELATE Scheme

In this section, we present an efficient scheme called RELATE as a solution for the problem defined in Definition 4.4. In short, RELATE examines the search space by starting from the input query Q_I. Then, it recursively *refines* it to obtain the next candidate queries, as explained earlier in Section 4.2.2.

The order in which RELATE visits the next query is determined by the traditional traversal strategies Breadth First (BFS) and Depth First Strategies (DFS). Employing BFS and DFS enables RELATE to incrementally compute M, i.e., incrementally computing the pairwise correlation of every candidate query, which leads to considerable cost savings.

Further, RELATE applies two simple yet powerful pruning techniques to enable far more efficient processing of the search space. These pruning techniques enable RELATE to avoid processing unqualified queries and to early abandon the correlation computations of unpromising pairs in M.

As shown in Figure 4.4, RELATE starts by the input query Q_I then recursively applies four refinement operations on the current query to obtain the next set of candidate queries. For each candidate query Q, RELATE has to compute the correlation of all pairs (i.e., M) within Q's interval. In the following subsection, we carefully examine the costs of computing a single matrix M.

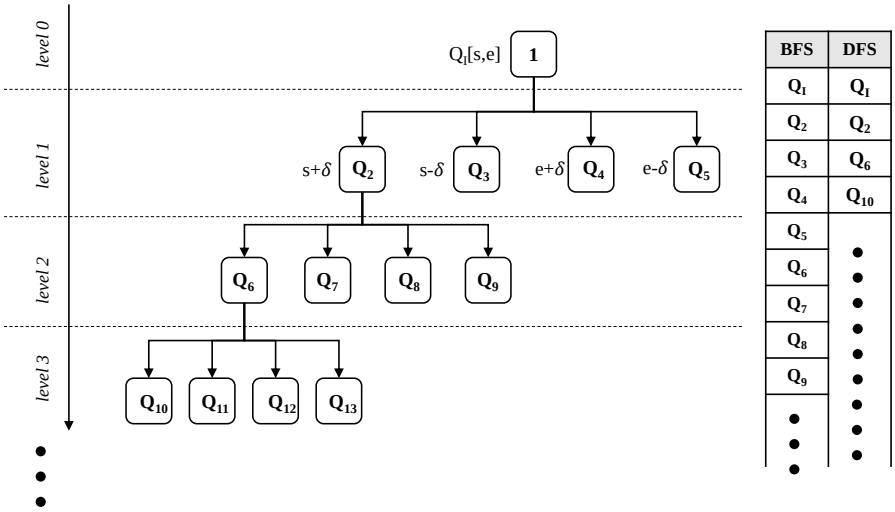

Figure 4.4: The classical traversal strategies: Breadth First (BFS) and Depth First (DFS) to decide the visiting order of the candidate queries in the search space starting from the input query Q_I

4.3.1 Cost Model Analysis

As shown in Figure 4.3 above, the computational cost of M dominates the I/O cost when there are thousands of time series. Hence, we focus on the computational bound costs involved when searching for Q^*. Moreover, we consider the I/O cost it takes to find Q^*.

1. **Number of Operations (OP):** OP is the number of operations to compute a correlation matrix M, and it depends on the length ℓ and the number n of the time series in Q. Specifically, for each pair of time series T_i, T_j of length ℓ in M, the five summation components in $\rho(T_i, T_j)$ will require exactly $(\ell-1)+(\ell-1)+2(\ell-1)+2(\ell-1)+2(\ell-1) = 8(\ell-1)$ operations. Hence, M requires a total of $\frac{n(n-1)}{2} \times 8(\ell-1)$ operations. Consequently, since there are candidate queries as many as $m(m-1)/2$, finding Q^* requires $m(m-1)/2 \times \frac{8n(n-1)(\ell-1)}{2}$ operations.

2. **Amount of Data Read (KBs):** As for the I/O cost, we define KBs to be the amount of data (in KBytes) an algorithm has to read to find Q^*. For a candidate query Q with length ℓ over $n = |R|$ time series, there are ℓn values to be read to compute M_Q. Assuming that a single value takes 8 bytes by default, then $\frac{8\ell n}{1024}$ KBs are required to compute M_Q. Thus, the total KBs with $m(m-1)/2$ candidate queries is $m(m-1)/2 \times \frac{8\ell n}{1024}$ KBs.

Note that, in practice, the number of data pages read from disk is a more common metric than the amount of data read. With the assumed storage layout in section 4.2.1, we define B as the size of a single page, such that $B \leq \frac{8n}{1024}$. As an example, the number of pages to be read for a query Q with length ℓ over $n = |R|$ time series is almost 2ℓ pages, where $B = 4$ KBs and $n = 1000$. In the rest of this chapter, we assume that $B = \frac{8n}{1024}$ for $n \geq 500$, i.e., each page read from disk corresponds to

the values of all time series for a specific time stamp. This sensible assumption simplifies the next discussion of RELATE optimization techniques and their effect on the costs.

The mainstream approaches in computing correlation efficiently [107, 70, 71, 128, 100] utilize the workload overlap to reduce the computational and I/O costs. It is clear from Figure 4.4 that candidate queries exhibit natural and rich overlap among them as well. Thus, similar to these approaches, RELATE utilizes this overlap to reduce the computational and I/O costs while searching for Q^*, as we explain next.

4.3.2 Caching Essential Arrays

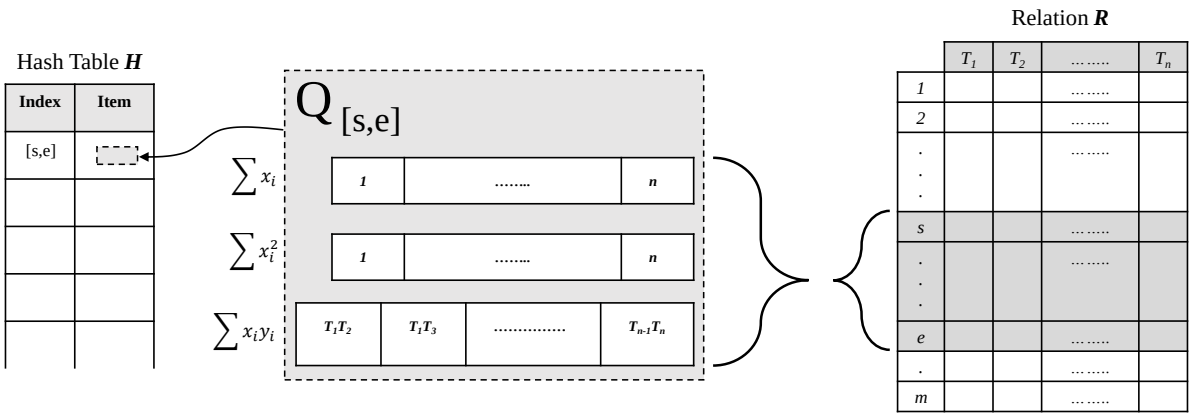

Figure 4.5: Essential arrays of a query $Q[s,e]$: Caching $\sum x$, $\sum x^2$ and $\sum xy$ of Q enables RELATE to incrementally compute the pairwise correlation of any pair in M for Q's offspring

Based on the observation that Eq. 4.2.1 can be computed incrementally [107], we propose in RELATE to cache the *essential arrays* $\sum x$, $\sum x^2$ and $\sum xy$ of a query after its correlation matrix has been evaluated and only if it happened to have offspring. This enables RELATE to reuse computations when computing M for Q's offspring later on, and lead to huge costs savings as explained in the following subsection.

The essential arrays of a query Q are added to memory by storing them into a simple data structure (e.g., a hash table) called H, indexed by Q's time interval. As visualized in Figure 4.5, each component is a 1-dimensional array of size n, except $\sum xy$ which is a 2-dimensional array of size $n \times n$. Alternatively, $\sum xy$ can be represented as a 1-dimensional array of size $n(n-1)/2$, where an element $\sum xy[i][j]$ is mapped to this 1-dimensional array as follows: $\sum xy[i*n+j]$. An element $\sum x[i]$ in $\sum x$ of Q denotes the sum of values of time series T_i that are within Q's time sub-interval. This convention is also followed for $\sum x^2[i]$ and $\sum xy[i][j]$.

4.3.3 Reusing Essential Arrays

RELATE reuses the stored arrays in H to avoid computing the pairwise correlation from scratch for every candidate query. Effectively, RELATE reduces the number of operations in $\rho(T_i, T_j)$ from $8(\ell - 1)$ to a constant number of operations that is independent of the length ℓ, for all time series pairs, as explained next.

Assume that RELATE has evaluated $Q[s,e]$ and stored its essential arrays in H. Further, assume the next candidate query $Q'[s-1,e]$ is generated by expanding the sub-interval $[s,e]$ by one value from the left hand side. Hence, $\ell' = \ell + 1$, where ℓ and ℓ' are the lengths of Q and Q', respectively.

It is clear that computing the pairwise correlation of each pair in Q requires $8(\ell - 1)$ operations. However, for any pair in Q', its pairwise correlation can be computed incrementally from Q's essential arrays that are stored in H. For instance, to compute the sum of values array $\sum x$ in Q', only the new values that are added to Q' (the values with time stamp $s-1$) are added to the sum of values array $\sum x$ of Q. Recall that from Eq. 4.2.1, computing the pairwise correlation $\rho(T_i, T_j)$ requires three arrays: $\sum x$, $\sum x^2$, and $\sum xy$, which all can be acquired incrementally with the same steps above. Thus, computing the pairwise correlation of any pair in Q' becomes independent of ℓ' and requires only updating the essential arrays by one value.

Maximum Size of Required Memory (MaxMemory): With the assumption of a limited space for caching the essential arrays, it is crucial for RELATE to minimize the size of H. Hence, RELATE should release a query from H once it is expired, i.e., its cached arrays will not be needed anymore. Thus, we consider the maximum size of H (MaxMemory) as a third cost metric, along with OP and KBs defined earlier.

In the following section, we explain how RELATE minimizes H's size by following two traditional traversal strategies while evaluating the candidate queries. Then, we introduce our proposed optimization techniques which further reduce the costs incurred while searching for the optimal solution.

4.3.4 Breadth-First and Depth-First Search Strategies

As we have mentioned previously, RELATE follows the traditional BFS and DFS traversal strategies to visit the candidate queries. These strategies enable RELATE to utilize and reuse the pairwise correlation computations across the candidate queries, and to minimize the size of required memory H, as we see next.

Algorithm 4.1 shows the main steps of the RELATE scheme. Users submit their input query Q_I,

Algorithm 4.1 RELATE

Input: Input query $Q_I[s,e]$, similarity weight λ, target correlation M_t
Output: Q^*, f_b

```
1:  f_b = -∞; Q* = φ; S = φ; H = φ; prevParent = Q_I;
2:  S.add(Q_I);
3:  while ( S ≠ φ ) do
4:      Q = S.pop();
5:      M_Q = ComputeAllPairwsie(Q,H);
6:      f = λP(Q_I,Q) + (1 − λ)C(M_Q,M_t);
7:      if ( f > f_b) then
8:          f_b = f; Q* = Q;
9:      C =refine(Q);
10:     for all q ∈ C do
11:         S.push(q);
12:     release(prev, prevParent,Q,H);
13:     prev = Q;
14:     prevParent = Q.parent;
15: return Q*, f_b;
```

the similarity weight λ, and the target correlation matrix M_t. The goal of RELATE is to iteratively refine Q_I until its results satisfies M_t, i.e., find the optimal refined query Q^*. Once found, RELATE outputs Q^* with the maximum (best) objective function f_b.

Initially, RELATE adds the input query Q_I to a data structure called S. Depending on the traversal strategy RELATE is assigned to use, S can be either FIFO queue (BFS), or a traditional stack (DFS). RELATE then enters a while loop until there are no more candidate queries in S. For each candidate query Q, RELATE calls the function $ComputeAllPairwise()$ to compute the current query's correlation matrix M_Q (more details will shortly follow). As RELATE progresses, it keeps track of Q^* and the minimum f which we call f_b in Algorithm 4.1 (lines 6-8). To generate the next set of candidate queries, RELATE refines Q's time interval (line 9) to have four refined queries, which are pushed into S as long as they have never been visited.

As listed in Algorithm 4.2, the $ComputeAllPairwise()$ function iterates over all pairs in M_Q. For a pair p, it searches H for an overlap p_o (i.e., essential arrays) based on Q's interval. If p_o is not found, then the essential arrays for p are computed from scratch by reading all values within Q's interval for that particular p. Otherwise, if p_o is found, its essential arrays are merged with the new values of p, and the pairwise correlation of p is computed incrementally, as discussed above in Section 4.3.3. In either cases, the algorithm updates H and adds p's essential arrays for later pairs and queries.

Theoretical Analysis: In this subsection we analyze the convergence and the correctness of the RELATE algorithm. The RELATE algorithm listed above returns an optimal query Q^* with the maximum objective value f_b among all possible candidate queries that are generated recursively

Algorithm 4.2 Compute Correlation Matrix

```
1:  procedure COMPUTEALLPAIRWISE(Q, H)
2:      for (i = 1 to n − 1) do
3:          for (j = i + 1 to n) do
4:              p = (i, j);                                          ▷ current pair
5:              p_o = H.findOverlap(p, Q);
6:              if ( p_o = φ ) then
7:                  fully probe(Q) for current pair p
8:              else
9:                  merge(p_o) with partially probed Q for p
10:             M_Q[i][j] = pairwise correlation for p, Eq. 4.2.1
11:             update H with p;
12:     return M_Q;
```

starting from an input query Q_I.

The algorithm converges when S (a FIFO queue or a stack) becomes empty. At each iteration, a query Q is evicted from S (initially, there is only one query, the input query Q_I, in S). Then, candidate queries are added to S by refining Q on the `timestamp` attribute, which result into a maximum of four candidate queries C. Each query $q \in C$ that has not been visited before is pushed into S. This process is continued recursively until S becomes empty. Note that the maximum number of candidate queries which the algorithm will visit for any given input query is $\frac{m(m-1)}{2}$, where m is the length of the time series in the dataset. This proves the convergence of the RELATE algorithm. Next, we prove the correctness and completeness of RELATE.

As discussed above, for any input Q_I and M_t, RELATE returns the optimal refined query Q^* with the maximum objective value f_b among all possible candidate queries. In the algorithm, Q^* is initialized to empty and $f_b = -\infty$. As RELATE iterates over all possible queries using the four refinement operations, Q^* and f_b are updated once a query is found to have a maximum f than f_b. Once the algorithm converges (i.e., terminates), Q^* and f_b will hold the optimal refined query with the maximum objective value among all possible candidate queries. As noted before, setting $\delta = 1$ ensures no possible candidate queries are missed.

Management of Cached Arrays

To minimize the size of H, RELATE removes cached arrays for a query once it has no benefit in the future. This depends on the traversal strategy that RELATE uses. We firstly discuss the case with BFS, then later we discuss how RELATE manages the cached arrays when it is using the DFS strategy. The pseudocode for both cases is listed in Algorithm 4.3.

Looking back at Figure 4.4, query Q_2's essential arrays will no longer be needed once RELATE

moves to evaluate query Q_3's offspring at level two (after evaluating Q_2' offspring, per the BFS strategy). To know which query is no longer needed, RELATE keeps track of the previous parent while evaluating candidate queries. Once it finishes evaluating a candidate, it invokes the $release()$ function which in turn checks if the current candidate's parent is different to the previous one. If this is the case, then RELATE removes the previous parent's essential arrays (e.g., query Q_2 in this case) since it is no longer needed.

With the DFS strategy, a query's essential arrays are needed as long as its offspring has not been fully evaluated. The reason is, RELATE requires the parent query of a given query to incrementally compute its correlation matrix. If RELATE releases this parent query's essential arrays before evaluating all of its offspring, then the unevaluated queries of its offspring will have to be computed from scratch.

Similar to the BFS strategy, RELATE invokes the $release()$ function once it finishes evaluating a candidate query. Under the DFS strategy, RELATE keeps adding the essential arrays as long as the current query's level is more than the previous one. As shown in Algorithm 4.3, once RELATE visits a query that is at a higher level, it recursively removes the essential arrays of all the queries starting from the previous query up to the parent of the current one. This is because when RELATE visits a query that is at a higher level than the previous one, it means that the previous query either has no offspring, or all queries that belong to the previous query's offspring have been evaluated. Hence, its essential arrays are no longer needed.

Effectively, RELATE reduces the costs metrics defined above by utilizing one of the two traversal strategies BFS and DFS in search for Q^*, and continuously releasing cached arrays once they become expired. Next, we further optimize RELATE by proposing two efficient and effective pruning techniques that provide further reductions of costs.

Algorithm 4.3 Release from Cache H

```
1:  procedure RELEASE(prev, prevParent, Q, H)
2:      switch Strategy do
3:          case BFS
4:              currParent = Q.Parent;
5:              if ( prevParent ≠ currParent ) then
6:                  remove prevParent's essential arrays from H
7:          case DFS
8:              currLevel = Q.level; prevLevel = prev.level;
9:              if ( (currLevel − prevLevel) < 0) then
10:                 recursively remove all essential arrays from H for queries starting from prev up to
    Q.Parent
```

4.3.5 Two-level Pruning Techniques

We extend RELATE scheme to utilize a monotonically decreasing property at two levels: similarity level, and pairwise correlation level. The former enables efficient pruning of the search space (i.e., pruning unpromising candidate queries), which addresses the quadratic number of candidate queries $\frac{m(m-1)}{2}$, while the latter addresses the quadratic number of pairs in a correlation matrix instance $\frac{n(n-1)}{2}$ by abandoning the computation of correlation for pairs as early as possible.

In the following subsections, we explain how these two techniques work, and discuss their mixed impact on the costs metrics we defined earlier in Section 4.3.1.

4.3.6 Similarity-aware Pruning Technique

RELATE applies a simple yet powerful pruning technique to avoid visiting unpromising queries. This result into a considerable computational and I/O costs savings.

Recall that the similarity function $S()$ in Eq. 4.2.2 computes a query's similarity to the initial time sub-interval defined in Q_I. This technique makes use of the monotonic property of the similarity function $S()$ to prune unpromising candidate queries that are far from Q_I.

Lemma 2. *$S()$ is a monotonically decreasing function.*

Proof. It is easy to see from Figure 4.4 that all candidate queries at level l have the same $S()$ since they are at the same distance from the root, i.e. $d(Q_I, Q_i)$. Also, it is clear that candidate queries at level $l+1$ are at one extra step from the queries in the previous level, hence, their $S()$ is lower. This pattern continues through out the whole tree. Hence, $S()$ is a monotonically decreasing function in terms of levels. ■

With Lemma 2, RELATE is able to early terminate the search and abandon all candidate queries that are yet to be explored once the current query's estimated $f()$ (i.e., f_e) is worse than the best solution found so far f_b. Specifically, if the current query's estimated objective $f_e = \lambda S() + (1 - \lambda) \times 1$ is lower than f_b, then this query, its offspring, the remaining queries in the queue, and all other unvisited queries will definitely have lower f than f_b and can no longer increase f_b further, thus they can be abandoned.

The impact of this technique on the costs is evident. Essentially, pruning candidate queries reduces the number of possible queries that RELATE has to visit, i.e., $\frac{m(m-1)}{2}$. As a result, the costs OP, KBs and MaxMemory are also reduced. However, we note that this reduction is controlled by the weight λ assigned to $S()$. Setting $\lambda = 0$ means this technique can no longer prune candidate queries, since $f_e = \lambda S() + (1 - \lambda) \times 1$ will always be higher than f_b.

4.3.7 Pairwise Correlation Pruning Technique

As mentioned above, setting the weight λ to zero prevents RELATE from pruning queries using the Similarity-aware Pruning Technique. Hence, we propose for RELATE to utilize a monotonically decreasing property of the correlation function $C()$ in Eq. 4.2.2 to early abandon the correlation computations of some pairs in M.

Lemma 3. *$C()$ is a monotonically decreasing function.*

Proof. Recall that $C(M_t, M_Q)$ is the normalized sum square of all absolute differences of pairwise correlation values between a target matrix M_t and a given one M_Q. By assuming M_Q to be an exact replica of M_t (which returns the maximum value of $C(M_t, M_Q)$), and iterating over all pairs in M_Q and inserting their real values, $C(M_t, M_Q)$ will gradually decrease. Specifically, assume the first half of pairs in M_Q have been probed (from $M_Q[0][1]$ to $M_Q[\frac{n-1}{2}][n]$), and the current value of $C(M_t, M_Q)$ equals to C_1:

$$C_1 = 1 - (\frac{1}{z} \sum_{i=0}^{\frac{n-1}{2}} \sum_{j=i+1}^{n} (M_t[i][j] - M_{Q^*}[i][j])^2)$$

The current value C_1 holds the differences of the first $n_1 = \frac{2n(n-1)}{2}$ pairs between M_Q and M_t, the remaining pairs will have zero difference by definition. The algorithm will continue probing the pairs $n_1 + 1$ to $\frac{n(n-1)}{2}$ to fully evaluate $C(M_t, M_Q)$. For instant, the value of the pair $n_1 + 1$ is computed as the squared difference between $M_Q[\frac{n-1}{2} + 1][\frac{n-1}{2} + 2]$ and $M_t[\frac{n-1}{2} + 1][\frac{n-1}{2} + 2]$ then it will replace the default value of zero in $C(M_t, M_Q)$. Let C_2 be the differences of the first n_1 pairs plus this new pair:

$$C_2 = 1 - (\frac{1}{z} \sum_{i=0}^{\frac{n-1}{2}} \sum_{j=i+1}^{n} (M_t[i][j] - M_{Q^*}[i][j])^2 + (M_Q[\frac{n-1}{2} + 1][\frac{n-1}{2} + 2] - M_t[\frac{n-1}{2} + 1][\frac{n-1}{2} + 2])^2)$$

Clearly, $C_2 \leq C_1$. Hence, $C(M_t, M_Q)$ is a monotonically decreasing function. ■

For a candidate instance Q, RELATE assumes that M_Q is an exact replica of M_t, then whenever the algorithm inserts a real correlation value for a pair in M_Q, it simultaneously checks if $f_e = \lambda P() + (1 - \lambda) C(M_t, M_Q)$ is lower than f_b. If so, the rest of the unevaluated pairs are skipped and the algorithm moves on to the next candidate query. For instance, assume that there are $n \times n$ pairs in M_Q for a query Q. According to Eq. 4.2.4, the value of $C(M_t, M_Q) = 1$ since M_Q is initially a replica of M_t. When evaluating the first pair, i.e., $M_Q[0][1]$, its real value difference from $M_t[0][1]$ is equal to or greater than the assumed value, hence, the amount $C(M_t, M_Q)$ will decrease as long as M_Q is populated with the real value of the $n \times n$ pairs.

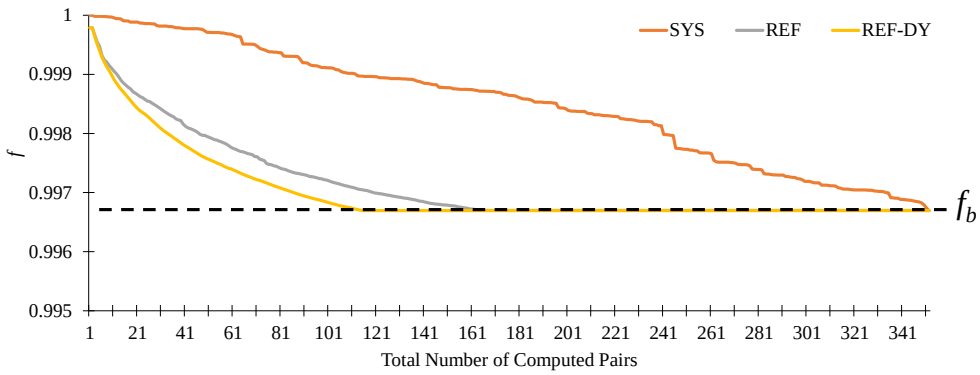

Figure 4.6: Ordering of pairs in a correlation matrix for a given candidate query. REF-DY is faster to arrive to f_b than SYS and REF, hence, enabling RELATE to reduce the computational cost

Due to the assumed storage layout, this technique is limited in minimizing the I/O cost, i.e., amount of data read (KBs). As mentioned above in Section 4.3.1, a single page of size B contains values of all time series for a specific time stamp. Hence, for a given correlation matrix M_Q, even if RELATE skips some pairs in M_Q and does not evaluate them, it has already read the values for all time series for a specific time stamp. As for the computational cost (OP) and the maximum memory required to cache the essential arrays (MaxMemory), this technique enables RELATE to minimize both costs since pruning a single pair corresponds to avoiding computing its pairwise correlation and storing its essential arrays in H.

The order which RELATE follows in examining the pairs in M_Q is crucial. Ideally, RELATE should follow an ordering that enables more pruning of pairs in M_Q to minimize the incurred costs.

4.3.8 Paris Ordering

As noted above, the order in which RELATE follows while evaluating the pairs in M_Q has an effect on the incurred costs. Specifically, RELATE should give a priority to evaluate a pair over another based on which one will bring the estimated f_e closer to the threshold f_b. The reason is, when RELATE evaluates the pairs in M_Q for a given candidate query, it can early terminate the computations of correlation for the pairs in M_Q once the estimated f_e becomes less than f_b.

We start by the default ordering of pairs, then propose two different orderings which enable RELATE to reduce the computational cost.

Systematic Ordering

The default ordering of pairs in RELATE is a systematic ordering (SYS). As its name suggests, SYS ordering is as follows:

$$M_Q[i][j] | i = 1,2,...,n-1; j = i+1, i+2, ..., n; i < j$$

Hence, RELATE examines the pairs for all candidate queries in the same exact order, and stops once f_e becomes less than f_b or all pairs in M_Q have been examined.

Greedy Ordering

Another more intuitive ordering is to rearrange the pairs in M_Q in an ascending order based on their scores. Informally, the score of a pair $M_Q[i][j]$ is its distance d to the corresponding pair in M_t. Formally:

$$score(M_Q[i][j]) = d(M_Q[i][j], M_t[i][j]) = |M_Q[i][j] - M_t[i][j]|$$

Accordingly, the first pair to be examined under this greedy ordering is the one with the maximum score. The intuition behind this ordering is to increase the chances of hitting the threshold f_b to early abandon the computations of correlation of the remaining pairs in M_Q.

Assume that M_Q has only two pairs $M_Q[i][j]$ and $M_Q[i'][j']$ with scores c' and c'', respectively, where $c'' > c'$. Further, assume $f_e = f_b + c$ (before evaluating either of the pairs) such that f_b and f_e are in the range (0-1) and $c'' > c > c'$. Clearly, evaluating the second pair $M_Q[i'][j']$ will lead to pruning the evaluation of the first pair, since f_e will be less than f_b. Conversely, evaluating the pair $M_Q[i][j]$ will not lead to pruning the evaluation of the second pair.

Since the values of the pairs in M_Q are not known until they are examined and evaluated, it becomes very difficult to acquire the scores without evaluating the pairs in the first place. Hence, RELATE reuses the computations and utilizes the history of the computed correlation values to estimate the scores of the pairs. Given a candidate query Q, RELATE exploits a reference matrix M_r that approximates the pairs' values in M_Q, and computes the distance d from that reference.

There are two types of a reference matrix M_r which RELATE can utilize for any given query Q:

- Static reference matrix (REF).
- Dynamic reference matrix (REF-DY).

In the static case, M_r is an exact copy of the input query's correlation matrix M_{Q_l}. This matrix is firstly computed by RELATE at the beginning of the algorithm. Consequently, RELATE computes the

order only once (after computing M_{Q_I}), then reuses this ordering for all remaining candidate queries.

In the dynamic case, M_r is an exact copy of the current query's parent matrix. Hence, RELATE re-computes the order every time the parent of a current query changes, which entails further cost overhead when compared to REF and SYS.

As illustrated in Figure 4.6, the two versions of greedy ordering (REF and REF-DY) enable REALTE to to early abandon the calculations of unnecessary pairs for a given query. For the case of static reference matrix, the pairs in M_{Q_I} that have the highest difference to the target matrix M_t are evaluated first, resulting in hitting the threshold (i.e., best solution found so far f_b) very early when compared to the systematic ordering.

4.4 Experiments

We have performed extensive experiments to evaluate the performance of our proposed algorithms. Before discussing the results, we first explain the experimental setup.

4.4.1 Setup

We experimented with the variants of RELATE scheme, as outlined in Table 4.2. Each traversal strategy is tested with the two pruning techniques proposed in Section 4.3.5. We also compare our algorithms with ZES [71], state-of-the-art algorithm proposed to discover the time sub-interval of a correlated pair with correlation above a threshold. Note that we extended ZES to cater for multi-variant analysis, i.e., a correlation matrix M for more than two time series. All algorithms have the cache management capability, as indicated in Table 4.2. The algorithms were implemented using Java SDK and run on a Windows machine with 16 GB RAM and Intel i7 CPU 3.0 GHz.

Datasets: In our experiments, we used two datasets: synthetic and a real dataset. The synthetic dataset was generated according to a Random Walk model, while the real dataset was extracted from Google Cluster Usage Data [103]. To generate a time series using the Random Walk model [39], a seed is set to a normally generated random number. Then, each subsequent number is the sum of its predecessor and a normally distributed number. There are a total of $n = 1000$ time series, and their maximum length is $m = 1000$.

Workload: a workload consists of a set of runs. Each run is a trio: input query, target correlation matrix and a user preference (Q_I, M_t, λ). A query's time interval $[s, e]$ length is either: short, medium or long. The interval is also either on the left hand side, right hand side, or in the middle of the original time series interval. Q_I is generated at random from these $2^3 + 1$ classifications. M_t is

Algorithm	**INC**	**SMP**	**PWC**	**Caching**
State-of-the-art (ZES)	✓	X	X	✓
RELATE-**BFS**-SMP	✓	✓	X	✓
RELATE-**BFS**-PWC	✓	X	✓	✓
RELATE-**BFS**-SMP-PWC	✓	✓	✓	✓
RELATE-**DFS**-SMP	✓	✓	X	✓
RELATE-**DFS**-PWC	✓	X	✓	✓
RELATE-**DFS**-SMP-PWC	✓	✓	✓	✓

Table 4.2: Variants of RELATE: INC: incremental computation of correlation. SMP: Similarity-aware pruning. PWC: Pairwise correlation pruning

arbitrarily chosen from a query that is generated from the above classifications to guarantee an exact solution existence. Hence, the size of the workload is $(2^3+1)^2$.

4.4.2 Results

We test the scalability of our proposed algorithms in terms of number and length of time series (n and m), and their sensitivity to the similarity weight λ. Moreover, we test the ordering methods SYS, REF and REF-DY of RELATE-PWC and their effects on the costs metrics. We report the cost components defined in Sections 4.3.1 and 4.3.3, and vary the parameters n, m and λ in the experiments.

We also use an additional parameter *ml* to specify the acceptable minimum length of any candidate query. Any candidate query with a length less than *ml*% of m is pruned.

We can ensure a reasonable running time for the experiments using *ml* and the default values of n and m. In particular, the parameter *ml* implicitly controls the number of candidate queries that are examined by our algorithms. The parameters n and m are dependent on the dataset itself, and we experiment with different scales of datasets.

Scalability Results

We evaluate the scalability of our algorithms in terms of time series length m and number of time series n using different parameters settings.

First, using the synthetic dataset, we vary m in the range [100 – 1000] while setting $n = 50, \lambda = 0.005$ and $ml = 0.5$. Figures 4.7 and 4.8 report the number of operations (OP) for the two strategies BFS and DFS. As the figures show, the two variations of RELATE scheme (BFS-SMP-PWC and DFS-SMP-PWC) reduce the amount of required operations OP by almost 40% when compared

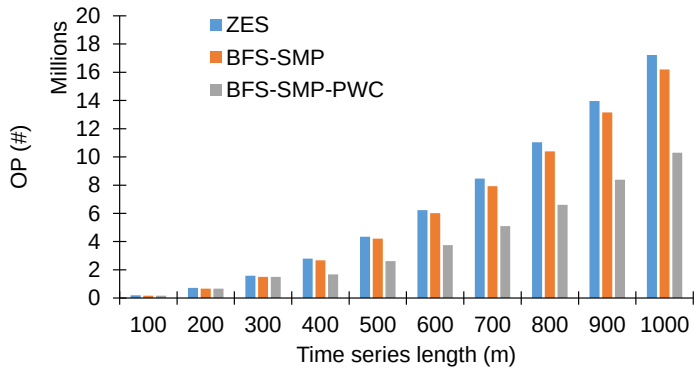

Figure 4.7: Average OP while varying time series length

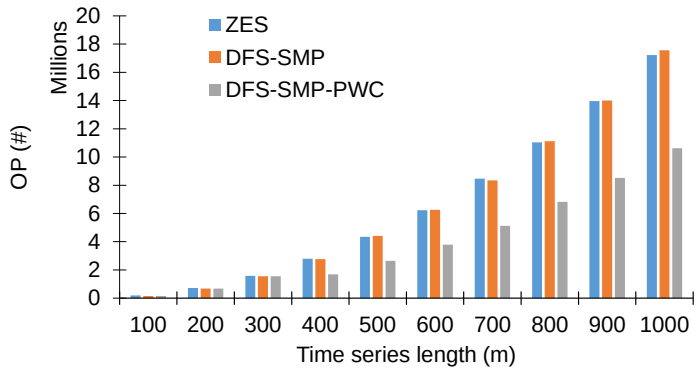

Figure 4.8: Average OP while varying time series length

to state-of-the-art algorithm ZES. This reduction is achieved by RELATE's pruning techniques. Specifically, both variations of RELATE scheme employe the pairwise correlation pruning technique which enable them to early abandon the pairwise correlation computations of some pairs of time series. Moreover, as Figure 4.7 shows, BFS-SMP is able to prune a small number of candidate queries by utilizing the similarity-aware pruning technique. However, the impact of this technique is limited because in this experiment λ is set to a small value while ml is set to a high value. That is, the former's pruning power decreases when it is set to a small value, while the latter's high value reduces the number of candidate queries which means less candidate queries to be pruned.

As for the amount of required memory to cache the essential arrays, we can see from Figures 4.9 and 4.10 that the DFS strategy is extremely poor in minimizing the amount of required memory, when compared to BFS and ZES. In contrast to BFS and ZES, DFS proceeds to evaluate Q's offsprings

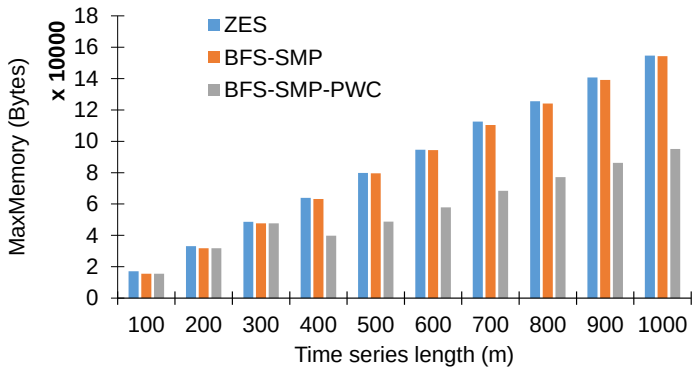

Figure 4.9: Average MaxMemory while varying time series length

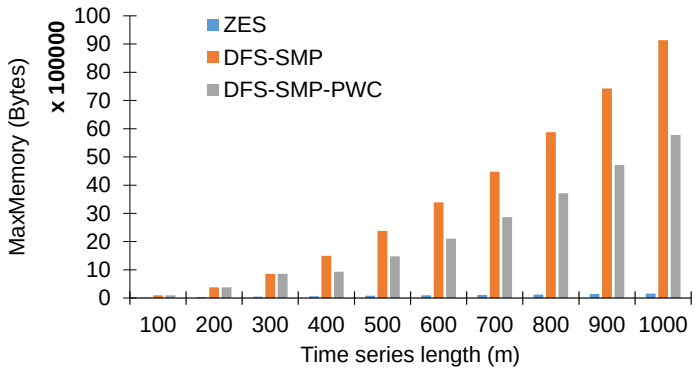

Figure 4.10: Average MaxMemory while varying time series length

before fully evaluating all Q parent's offspring. Hence, DFS needs to keep the essential arrays of Q and Q's parent in memory. As explained in Section 4.3.4, DFS needs both Q and Q's parent essential arrays to incrementally compute the pairwise correlation. The insertion of the essential arrays in H by DFS continues as long as there are offsprings, i.e., the level of the current query is more than the previous query. Thus, under the DFS strategy, H will have essential arrays for $m(m-1)/2$ candidate queries in the worst case.

However, the two algorithms BFS-SMP-PWC and ZES only proceed to evaluate Q's offspring after fully evaluating all Q parent's offspring. This enable both algorithms to reduce the amount of required memory by nearly 10 factors, as shown in Figures 4.9 and 4.10. Figure 4.9 shows that the pairwise correlation pruning technique can effectively reduce the amount of required memory by 39%, while the similarity-aware pruning technique's impact on MaxMemory is negligible since λ is

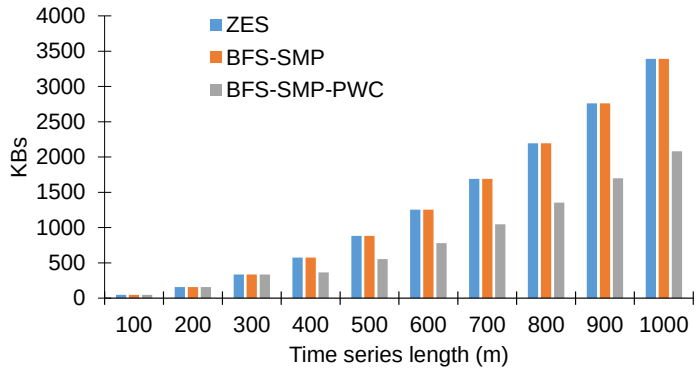

Figure 4.11: Average KBs while varying time series length

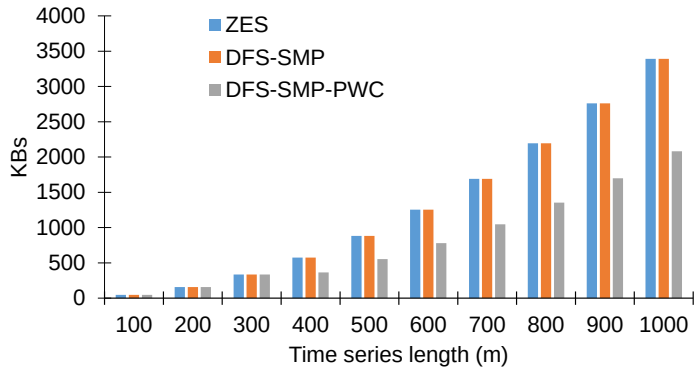

Figure 4.12: Average KBs while varying time series length

set to a small value.

Figures 4.11 and 4.12 illustrate the average I/O cost for RELATE scheme variations against ZES. By comparison to ZES, BFS-SMP-PWC and DFS-SMP-PWC are able to reduce the amount of I/O by up to 39%, mainly because of the pairwise correlation pruning technique. Again, the similarity-aware pruning technique shows a negligible impact since λ is set to a small value.

Second, we show in this experiment the results for the real dataset: Google Cluster Usage Data, while varying the number of time series n in the range $[10-100]$ and setting $m = 200, \lambda = 0.005, ml = 0.1$. Because BFS dominates DFS in the costs metrics, we only compare the BFS strategy to ZES in the remaining experiments.

As Figures 4.13, 4.14 and 4.15 show, all three cost metrics (computational cost, maximum memory required and amount of data read) increase as the number of time series increases. This

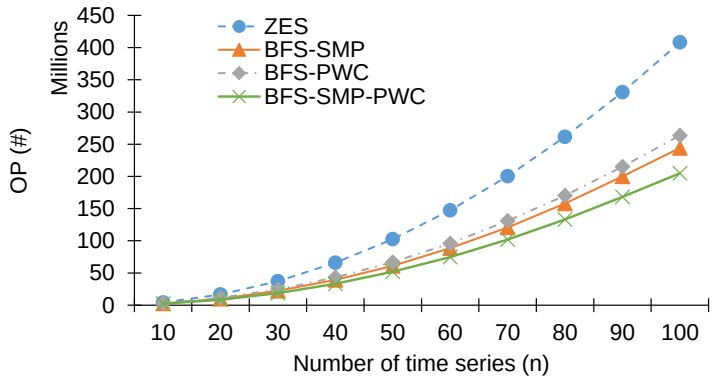

Figure 4.13: Average OP while varying number of time series

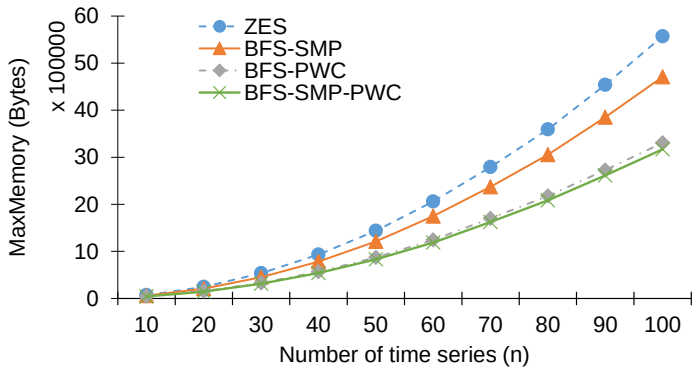

Figure 4.14: Average MaxMemory while varying number of time series

is because the number of pairs in a correlation matrix increases quadratically with the number of time series, as mentioned previously. Despite that, our proposed algorithm BFS-SMP-PWC is able to reduce the costs increase by 40%-50%, when compared to ZES. For instance, Figure 4.13 shows that the similarity-aware and pairwise correlation pruning techniques reduce the computational cost by 40% and 35%, respectively. Similarly, Figure 4.15 shows that the similarity-aware and pairwise correlation pruning techniques collectively reduce the amount of required memory by 41%.

Sensitivity to λ Results

In this experiment, we measure the sensitivity of λ on the costs metrics while setting $n = 20, m = 200$ and $ml = 0.1$ using the Google Cluster Usage Data. We compare the results of the BFS's variants to show the pruning techniques impact on the costs metrics for different settings of λ.

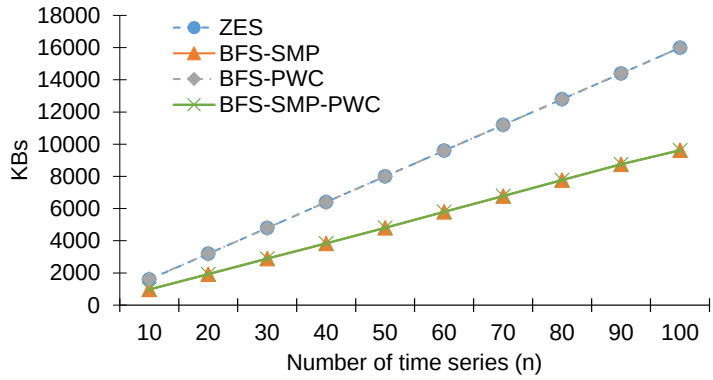

Figure 4.15: Average KBs while varying number of time series

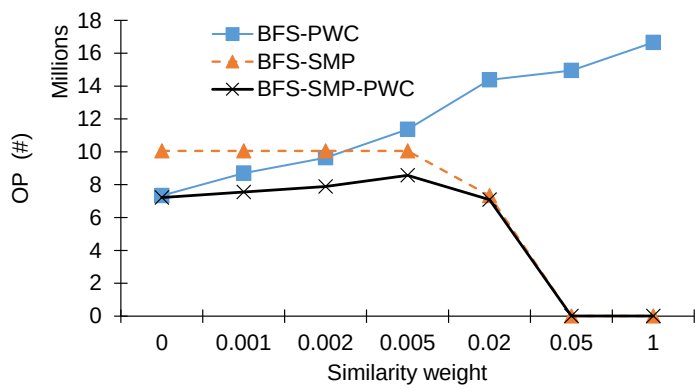

Figure 4.16: Average OP while varying similarity weight λ

Figures 4.16 and 4.17 show an interesting relationship between the similarity weight λ and the cost metrics. As mentioned previously, $S()$ is a monotonically decreasing function, i.e., it decreases as the algorithm moves away from the root (Q_I). The algorithm BFS-SMP-PWC utilizes this property to prune unqualified queries. This is apparent in both figures, as more weight is assigned to $S()$, i.e., increasing λ, BFS-SMP-PWC prunes more unqualified queries, resulting into reducing the number of operations and the amount of memory required to cache essential arrays. However, the BFS-PWC variant becomes less efficient in its pairwise correlation pruning technique, since the threshold f_b becomes more loose due to the less weight $C()$ gets as $S()$'s weight increases. The BFS-SMP-PWC variant which combines the two techniques achieves the maximum costs reduction, when compared to the other two variants BFS-SMP and BFS-PWC.

Figure 4.18 shows the amount of read data while increasing λ for the three variants BFS-PWC,

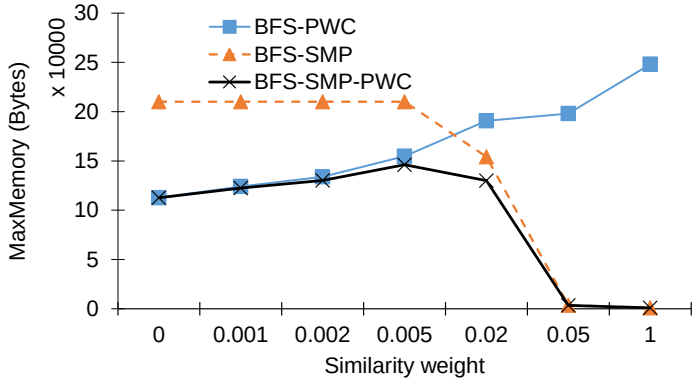

Figure 4.17: Average MaxMemory while varying similarity weight λ

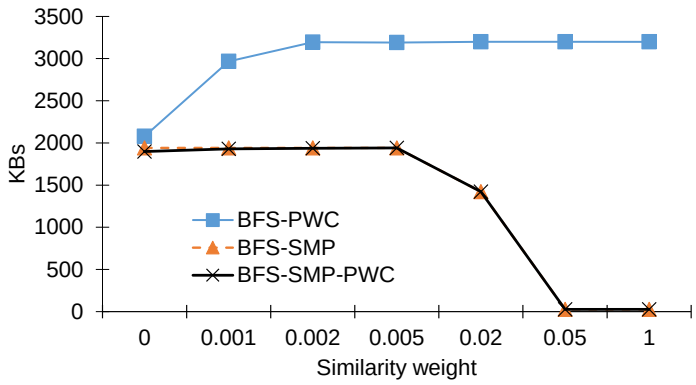

Figure 4.18: Average KBs while varying similarity weight λ

BFS-SMP and BFS-SMP-PWC. Clearly, the BFS-SMP-PWC variant achieves the maximum cost reduction as well, since it combines both pruning techniques.

Ordering of Pairs Results

As mentioned in Section 4.3.7, the order which RELATE follows in examining the pairs in M can have an effect on the overall performance. Using the Google Cluster Usage Dataset, we examine in this experiment how ordering of pairs can effect the cost metrics of our algorithms while varying n in the range [5 – 30] and setting $m = 200, \lambda = 0, ml = 0.1$. We report the results of the three different approaches of ordering: systematic (SYS), greedy (REF) and dynamic greedy (REF-DY), under the BFS strategy.

Figures 4.19 and 4.20 show the computational cost and number of probed pairs for the three

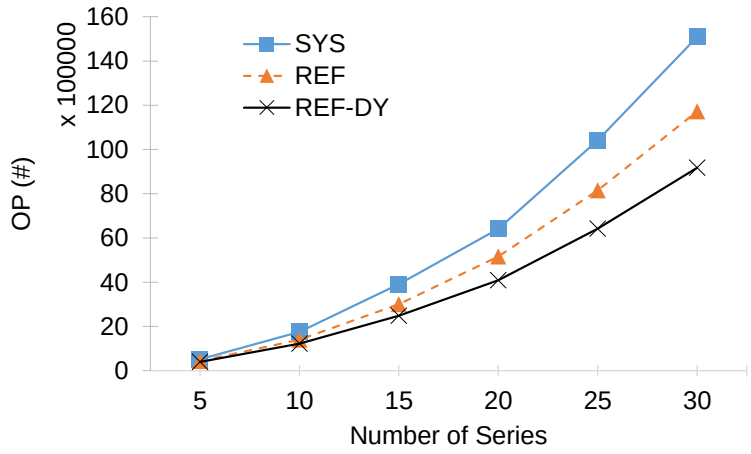

Figure 4.19: Computational cost across different ordering methods

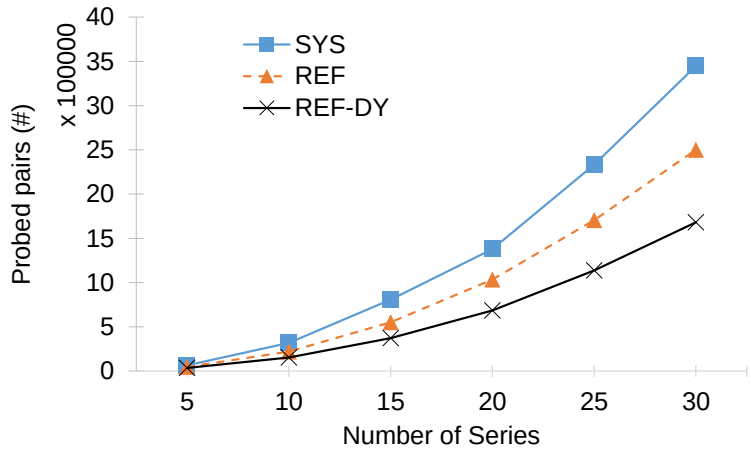

Figure 4.20: Number of probed pairs across different ordering methods

aforementioned approaches of ordering. Recall that REF-DY reorders the pairs when the current candidate query's parent changes, while REF performs this ordering once at the beginning based on the input query.

From the figures, the computational cost and the number of examined pairs can be further reduced by almost 40% and 52%, respectively, if RELATE uses the REF-DY method to order the pairs instead of the default ordering SYS. While this seems very promising, recomputing the distances when the parent of a candidate query changes entails further computational cost, as shown in Table 4.3. REF ordering provides a relatively competitive reduction of 22% and 30% for the computational cost and number of probed pairs, respectively, when compared to SYS. However, in contrast to REF-DY, REF incurs zero additional cost.

The average I/O cost for these approaches are relatively similar, as shown in Figure 4.21. Reducing the number of probed pairs implicitly means reducing the I/O cost as well, which is evident when looking at the two Figures 4.21 and 4.20.

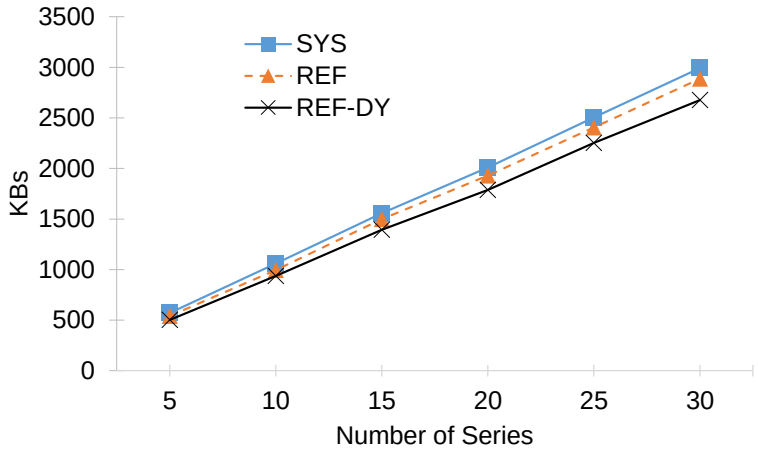

Figure 4.21: Amount of read data across different ordering methods

Number of series	5	10	15	20	25	30
Operations (#)	33788	153264	370228	680697	1131465	1671644

Table 4.3: Additional computational costs of REF-DY

Results Discussion

Compared to state-of-the-art, RELATE shows significant reduction in costs across all performance measurements while varying m (length of time series) and n (number of time series). The is due to the similarity-aware and pairwise correlation pruning techniques which enable RELATE to reuse computations and early terminate the search when possible. However, the DFS version of RELATE algorithm shows poor performance in terms of MaxMemory because of the traversal strategy that DFS implements. Overall, the BFS-SMP-PWC variant of RELATE outperforms the other variants across all performance measurements. All variants of RELATE are sensitive to the similarity weight λ. Increasing λ result in pruning more queries because of the similarity-aware pruning technique. However, as the similarity-aware pruning technique weight is increased, the pairwise correlation pruning technique becomes less efficient, hence, the BFS-PWC variant shows poor performance with increasing λ. Computational costs of RELATE are also sensitive to the ordering of pairs. A greedy ordering (REF) provides relatively competitive reductions in costs when compared to the other ordering methods.

4.5 Summary

Automatically refining a query to satisfy certain pairwise correlation constraints assists users in exploring time series data for various applications such as network and environment monitoring. However, finding a refined query that optimally satisfies pairwise correlation constraints entails

massive computational and I/O costs, since the number of possible candidate queries increases quadratically with the length of time series. Moreover, the number of time series pairs increases quadratically as well with the number of time series, making it impractical for brute-force algorithms.

Therefore, in this chapter we proposed an efficient scheme called RELATE which aims to refine a user's query to satisfy the pairwise correlation constraints for all time series pairs, while maximizing the similarity between the refined query and the initial one. Specifically, RELATE iteratively refines an input query's sub-interval and compute the pairwise correlation of all pairs of time series for each candidate query. At a high level, RELATE builds on the observation that pairwise correlation can be computed incrementally to save computational and I/O costs. To achieve that, RELATE employs two traditional traversal strategies: BFS and DFS, to order the evaluation of the candidate queries. These two strategies enable RELATE to control the incurred costs by utilizing the computed pairwise correlation of previous queries in an efficient way.

Beyond the incremental computation of pairwise correlation, we proposed two pruning techniques that are based on a decreasing monotonic property to address the computational and I/O costs. The first technique enables RELATE to prune unpromising candidate queries based on their similarity to the input query, while the second technique enables RELATE to early abandon the calculation of pairwise correlation for some time series pairs. Then, we evaluated our RELATE scheme under different experimental settings and compared its performance against state-of-the-art algorithm using both real and synthetic datasets.

CHAPTER 5

Conclusions

The aim of this thesis is to develop efficient query refinement techniques to guide users in refining their queries when exploring relational and sequential data. In Section 5.1 we reflect on the novel contributions in this thesis, and provide directions for future work in Section 5.2.

5.1 Summary of Contributions

In Chapter 3, we addressed the problem of refining a query given aggregate and similarity constraints. Initially, in Section 3.2 we presented preliminaries and the formal definition for the Similarity-aware, Aggregate-based Query Refinement problem. We also defined our query model and provided a declarative model to enable users to specify all relevant parameters for refining their queries in an extended SQL structure.

In Section 3.3, we focused on a special case of an aggregate constraint: cardinality constraint, for which we proposed a novel scheme called SAQR. SAQR balances the tradeoff between satisfying the cardinality and similarity constraints imposed on the refined query so that to maximize its overall benefit to the user. To achieve that goal, SAQR implements efficient strategies to minimize the costs incurred in exploring the available search space. In particular, SAQR utilizes both similarity- and cardinality-based pruning techniques to bound the search space and quickly find a refined query that meets the user expectations. Moreover, SAQR employs a hierarchical representation of the search space, which provides better estimates of the cardinality constraint bounds.

Then, in Section 3.4, we extended the cardinality constraint to include SQL's standard aggregate operators `sum`, `avg`, `min`, `max` as constraints for refinements, which is the case in the Similarity-aware, Aggregate-based Query Refinement problem. We proposed an efficient scheme for refinement of aggregate constraints, which extends on SAQR techniques to satisfy the aggregate and similarity constraints. Specifically, EAGER implements efficient strategies to minimize the

costs incurred in exploring the search space by utilizing similarity and the monotonicity property to bound the search space and quickly find a refined query that meets users expectations. Further, we proposed optimization and approximation techniques for EAGER scheme which offer a controllable tradeoff between the cost and accuracy of the refined queries. These techniques include strategic materialization of candidate queries and scoring candidate queries based on similarity and aggregate bounds.

In Section 3.5, we presented a web-based application (ORange) which employs the novel SAQR schemes to refine a query given cardinality and similarity constraint. The aim of ORange is to guide police coordinators in allocating service zones to police officers by refining an initial range query to satisfy a specific cardinality constraint. Initially, ORange presents users with a real map and allows them to select a service zone and also to enter a capacity (cardinality) constraint with other controlling parameters. After refining the selected zone, ORange shows the refined zone on the map and the performance results of each scheme to provide post-run evaluation of the system's capabilities.

Lastly, in Chapter 4 we addressed the problem of Similarity-aware, Correlation-based Query Refinement. This computationally hard problem has many applications in areas such as network and environment monitoring. We explained that brute force algorithms are not practical to solve such a problem since the number of possible sub-intervals (i.e., candidate refined queries) and the number of time series pairs grow quadratically with the length and number of time series. Exhaustively examining these candidate queries and time series pairs entails massive computational and I/O costs. Therefore, we proposed an efficient scheme called RELATE which aims to refine a user's query to satisfy pairwise correlation constraints while maximizing the similarity of the refined query to the initial one. To minimize the incurred costs, RELATE employs two traditional traversal strategies: BFS and DFS, to order the evaluation of the candidate queries. These two strategies enable RELATE to control the incurred costs by incrementally computing the pairwise correlation of the candidate queries.

Beyond the incremental computation of pairwise correlation, we proposed two pruning techniques that are based on a decreasing monotonic property to address the computational and I/O costs. The first technique enables RELATE to prune unpromising candidate queries based on their similarity to the input query, while the second technique enables RELATE to early abandon the calculation of pairwise correlation for some time series pairs.

5.2 Future Work

In this section we briefly discuss future work directions that are related to the research problems addressed within this thesis.

5.2.1 Query Refinement for Aggregate Constraints

We identified two leading directions to extend the aggregate-based query refinement problem. At the problem definition level, a query in Chapter 3 is represented by its range predicates which resembles a d-dimensional rectangle, such as the 2-dimensional service zone in the ORange demo in Section 3.5. Further, an aggregate constraint is applied on the result of a single aggregate operator over a single attribute. The above assumptions can be extended by considering complex queries and complex aggregate constraints.

For instance, spatial queries in the 2-dimensional space can be used in ORange to select a service zone that is not rectangular, e.g., a circular shape. Supporting such complex spatial queries introduces unique challenges and requires revisiting the defined refinement model comprehensively, starting from the predicates' forms, measuring the refined queries similarities, the refinement operations (i.e., predicates expansion and contraction), the enumeration of refined queries, the aggregates bounds (i.e., finding the dominating, and dominated by queries) and the hierarchical representation of the search space.

While a refinement constraint in Chapter 3 is assumed to be singular, i.e., consists of a single aggregate operator over a single attribute, it can be extended to support complex aggregate constraints. A complex aggregate constraint can be defined as the result of joining two singular aggregate constraints using a binary operation, e.g., a ratio, where the first constraint is divided by the second constraint. Such constraints require further investigation to estimate suitable bounds to enable bounds-based pruning techniques.

5.2.2 Query Refinement for Correlation Constraints

The addressed problem in Chapter 4 draws its hardness from the quadratic number of possible sub-intervals, and the quadratic number of pairs that need to be evaluated. The former depends on the length of the time series, while the latter depends on the number of time series.

Approximation techniques can be used to address this computationally hard problem. Although such techniques introduce inaccuracy, they provide a suitable trade off between query-time and accuracy which is favored in some data exploration applications.

As briefly noted in Section 4.2.2, one possible technique to control the quadratic number of possible sub-intervals is to set the step size δ to a larger value. Moreover, the time series data can be approximated offline to control the quadratic number of pairs. This can be done in two steps. First, using the classical moving average technique, the time series data are smoothed based on a defined interval. Then, in the second step, time series are grouped based on a similarity threshold. Each group is represented by an artificial time series that represents all the real time series within that group. However, this requires formally defining how pairwise correlation constraints are assigned to these approximated time series. For instance, two pairwise correlation constraints in M might correspond to a single approximated pair. Thus, such approximation techniques should address these inconsistent constraints in the approximated space.

References

[1] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah. Time-series clustering–a decade review. *Information Systems*, 53:16–38, 2015.

[2] A. Albarrak and M. A. Sharaf. Efficient schemes for similarity-aware refinement of aggregation queries. *World Wide Web*, pages 1–31, 2017.

[3] A. Albarrak, M. A. Sharaf, and X. Zhou. SAQR: an efficient scheme for similarity-aware query refinement. In *Database Systems for Advanced Applications - 19th International Conference, DASFAA 2014, Bali, Indonesia, April 21-24, 2014. Proceedings, Part I*, pages 110–125, 2014.

[4] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *PODS*, pages 265–272, 1990.

[5] P. Barceló and M. Romero. The complexity of reverse engineering problems for conjunctive queries. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, pages 7:1–7:17, 2017.

[6] J. E. Barlett, J. W. Kotrlik, and C. C. Higgins. Organizational research: Determining appropriate sample size in survey research. *Information technology, learning, and performance journal*, 19(1):43, 2001.

[7] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 131–140, 2007.

[8] C. R. Blyth. On simpson's paradox and the sure-thing principle. *Journal of the American Statistical Association*, 67(338):364–366, 1972.

[9] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, pages 421–430, 2001.

[10] M. Brucato, A. Abouzied, and A. Meliou. Improving package recommendations through query relaxation. In *Proceedings of the First International Workshop on Bringing the Value of "Big Data" to Users, Data4U@VLDB 2014, Hangzhou, China, September 1, 2014*, page 13, 2014.

[11] M. Brucato, R. Ramakrishna, A. Abouzied, and A. Meliou. Packagebuilder: From tuples to packages. *PVLDB*, 7(13):1593–1596, 2014.

[12] M. Brucato, J. F. Beltran, A. Abouzied, and A. Meliou. Scalable package queries in relational database systems. *PVLDB*, 9(7):576–587, 2016.

[13] N. Bruno and S. Chaudhuri. Flexible database generators. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1097–1107, 2005.

[14] N. Bruno, S. Chaudhuri, and D. Thomas. Generating queries with cardinality constraints for dbms testing. *IEEE Trans. Knowl. Data Eng.*, 18(12):1721–1725, 2006.

[15] U. Çetintemel, M. Cherniack, J. DeBrabant, Y. Diao, K. Dimitriadou, A. Kalinin, O. Papaemmanouil, and S. B. Zdonik. Query steering for interactive data exploration. In *CIDR*, 2013.

[16] A. Chapman and H. V. Jagadish. Why not? In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 523–534, 2009.

[17] S. Chaudhuri. Generalization and a framework for query modification. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 138–145, 1990.

[18] S. Chaudhuri. What next?: a half-dozen data management research goals for big data and the cloud. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–4, 2012.

[19] S. Chaudhuri and V. Narasayya. Program for generating skewed data distributions for tpc-d. ftp://ftp.research.microsoft.com/users/surajitc/TPCDSkew/.

[20] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 888–899, 2004.

[21] Y. Chen, W. Wang, X. Du, and X. Zhou. Continuously monitoring the correlations of massive discrete streams. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 1571–1576, 2011.

[22] Z. Chen, T. Li, and Y. Sun. A learning approach to SQL query results ranking using skyline and users' current navigational behavior. *IEEE Trans. Knowl. Data Eng.*, 25(12):2683–2693, 2013.

[23] C. J. Date. Database usability. In *SIGMOD'83, Proceedings of Annual Meeting, San Jose, California, May 23-26, 1983.*, page 1, 1983.

[24] Y. Diao, K. Dimitriadou, Z. Li, W. Liu, O. Papaemmanouil, K. Peng, and L. Peng. AIDE: an automatic user navigation system for interactive data exploration. *PVLDB*, 8(12):1964–1967, 2015.

[25] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. Explore-by-example: an automatic query steering framework for interactive data exploration. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 517–528, 2014.

[26] K. Dimitriadou, O. Papaemmanouil, and Y. Diao. AIDE: an active learning-based approach for interactive data exploration. *IEEE Trans. Knowl. Data Eng.*, 28(11):2842–2856, 2016.

[27] Y. Ding and E. J. Keogh. Query suggestion to allow intuitive interactive search in multidimensional time series. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*, pages 18:1–18:11, 2017.

[28] M. Drosou and E. Pitoura. Search result diversification. *SIGMOD Record*, 39(1):41–47, 2010.

[29] M. Drosou and E. Pitoura. Redrive: result-driven database exploration through recommendations. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 1547–1552, 2011.

[30] M. Drosou and E. Pitoura. Ymaldb: exploring relational databases via result-driven recommendations. *VLDB J.*, 22(6):849–874, 2013.

[31] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[32] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[33] J. Fan, G. Li, and L. Zhou. Interactive SQL query suggestion: Making databases user-friendly. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 351–362, 2011.

[34] K. Feng, G. Cong, S. S. Bhowmick, W. Peng, and C. Miao. Towards best region search for data exploration. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1055–1070, 2016.

[35] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[36] M. Gavrilov, D. Anguelov, P. Indyk, and R. Motwani. Mining the stock market (extended abstract): which measure is best? In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, Boston, MA, USA, August 20-23, 2000*, pages 487–496, 2000.

[37] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals. *Data Min. Knowl. Discov.*, 1(1):29–53, 1997.

[38] T. Guo, J. Calbimonte, H. Zhuang, and K. Aberer. Sigco: Mining significant correlations via a distributed real-time computation engine. In *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pages 747–756, 2015.

[39] T. Guo, S. Sathe, and K. Aberer. Fast distributed correlation discovery over streaming time-series data. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, pages 1161–1170, 2015.

[40] Y. Guo, C. Binnig, and T. Kraska. What you see is not what you get!: Detecting simpson's paradoxes during data exploration. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics, SIGMOD 2017, Chicago, IL, USA, May 14, 2017*, pages 2:1–2:5, 2017.

[41] J. F. Hair, W. C. Black, B. J. Babin, R. E. Anderson, R. L. Tatham, et al. *Multivariate data analysis*, volume 5. Prentice hall Upper Saddle River, NJ, 1998.

[42] P. Hanrahan. Analytic database technologies for a new kind of user: the data enthusiast. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 577–578, 2012.

[43] Z. He and E. Lo. Answering why-not questions on top-k queries. *IEEE Trans. Knowl. Data Eng.*, 26(6):1300–1315, 2014.

[44] K. Hose and A. Vlachou. A survey of skyline processing in highly distributed environments. *VLDB J.*, 21(3):359–384, 2012.

[45] S. Idreos. *Big Data Exploration*. Taylor and Francis, 2013.

[46] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of data exploration techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 277–281, 2015.

[47] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-*k* query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[48] M. S. Islam, C. Liu, and R. Zhou. On modeling query refinement by capturing user intent through feedback. In *Twenty-Third Australasian Database Conference, ADC 2012, Melbourne, Australia, January 2012*, pages 11–20, 2012.

[49] M. S. Islam, C. Liu, and R. Zhou. A framework for query refinement with user feedback. *Journal of Systems and Software*, 86(6):1580–1595, 2013.

[50] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 973–984, 2013.

[51] M. S. Islam, C. Liu, and J. Li. Efficient answering of why-not questions in similar graph matching. *IEEE Trans. Knowl. Data Eng.*, 27(10):2672–2686, 2015.

[52] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 13–24, 2007.

[53] D. Jannach. Techniques for fast query relaxation in content-based recommender systems. In *KI 2006: Advances in Artificial Intelligence, 29th Annual German Conference on AI, KI 2006, Bremen, Germany, June 14-17, 2006, Proceedings*, pages 49–63, 2006.

[54] L. Jiang and A. Nandi. Snaptoquery: Providing interactive feedback during exploratory query specification. *PVLDB*, 8(11):1250–1261, 2015.

[55] M. Joglekar, H. Garcia-Molina, and A. G. Parameswaran. Interactive data exploration with smart drill-down. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 906–917, 2016.

[56] R. A. Johnson, D. W. Wichern, et al. *Applied multivariate statistical analysis*, volume 4. Prentice-Hall New Jersey, 2014.

[57] A. Kadlag, A. V. Wanjari, J. Freire, and J. R. Haritsa. Supporting exploratory queries in databases. In *DASFAA*, pages 594–605, 2004.

[58] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Interactive data exploration using semantic windows. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 505–516, 2014.

[59] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Searchlight: Enabling integrated search and exploration over large multidimensional data. *PVLDB*, 8(10):1094–1105, 2015.

[60] V. Kantere. Query similarity for approximate query answering. In *Database and Expert Systems Applications - 27th International Conference, DEXA 2016, Porto, Portugal, September 5-8, 2016, Proceedings, Part II*, pages 355–367, 2016.

[61] V. Kantere, G. Orfanoudakis, A. Kementsietsidis, and T. K. Sellis. Query relaxation across heterogeneous data sources. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, pages 473–482, 2015.

[62] E. J. Keogh and S. Kasetty. On the need for time series data mining benchmarks: A survey and empirical demonstration. *Data Min. Knowl. Discov.*, 7(4):349–371, 2003.

[63] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011.

[64] A. Key, B. Howe, D. Perry, and C. R. Aragon. Vizdeck: self-organizing dashboards for visual analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 681–684, 2012.

[65] H. A. Khan, M. A. Sharaf, and A. Albarrak. Divide: efficient diversification for interactive data exploration. In *Conference on Scientific and Statistical Database Management, SSDBM '14, Aalborg, Denmark, June 30 - July 02, 2014*, pages 15:1–15:12, 2014.

[66] N. Koudas, C. Li, A. K. H. Tung, and R. Vernica. Relaxing join and selection queries. In *VLDB*, pages 199–210, 2006.

[67] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, pages 401–412, 2001.

[68] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. Lars: A location-aware recommender system. In *ICDE*, pages 450–461, 2012.

[69] H. Li, C. Chan, and D. Maier. Query from examples: An iterative, data-driven approach to query construction. *PVLDB*, 8(13):2158–2169, 2015.

[70] Y. Li, L. H. U, M. L. Yiu, and Z. Gong. Discovering longest-lasting correlation in sequence databases. *PVLDB*, 6(14):1666–1677, 2013.

[71] Y. Li, L. H. U, M. L. Yiu, and Z. Gong. Efficient discovery of longest-lasting correlation in sequence databases. *VLDB J.*, 25(6):767–790, 2016.

[72] J. Lin, E. J. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom. Visually mining and monitoring massive time series. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Seattle, Washington, USA, August 22-25, 2004*, pages 460–469, 2004.

[73] J. Liu and A. Terzis. Sensing data centres for energy efficiency. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 370(1958): 136–157, 2012.

[74] W. Luo, M. Gallagher, and J. Wiles. Parameter-free search of time-series discord. *J. Comput. Sci. Technol.*, 28(2):300–310, 2013.

[75] A. Marian, N. Bruno, and L. Gravano. Evaluating top-*k* queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2):319–362, 2004.

[76] Y. Matsubara, Y. Sakurai, N. Ueda, and M. Yoshikawa. Fast and exact monitoring of co-evolving data streams. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 390–399, 2014.

[77] D. Mindolin and J. Chomicki. Discovering relative importance of skyline attributes. *PVLDB*, 2(1):610–621, 2009.

[78] C. Mishra and N. Koudas. Interactive query refinement. In *EDBT*, pages 862–873, 2009.

[79] C. Mishra, N. Koudas, and C. Zuzarte. Generating targeted queries for database testing. In *SIGMOD Conference*, pages 499–510, 2008.

[80] S. Monchaux, F. Amadieu, A. Chevalier, and C. Mariné. Query strategies during information searching: Effects of prior domain knowledge and complexity of the information problems to be solved. *Inf. Process. Manage.*, 51(5):557–569, 2015.

[81] K. Morton, M. Balazinska, D. Grossman, and J. D. Mackinlay. Support the data enthusiast: Challenges for next-generation data-analysis systems. *PVLDB*, 7(6):453–456, 2014.

[82] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: Give me an example of what you need. *PVLDB*, 7(5):365–376, 2014.

[83] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. IQR: an interactive query relaxation system for the empty-answer problem. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1095–1098, 2014.

[84] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. Exemplar queries: a new way of searching. *VLDB J.*, 25(6):741–765, 2016.

[85] D. Mottin, A. Marascu, S. B. Roy, G. Das, T. Palpanas, and Y. Velegrakis. A holistic and principled approach for the empty-answer problem. *VLDB J.*, 25(4):597–622, 2016.

[86] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. New trends on exploratory methods for data analytics. *PVLDB*, 10(12):1977–1980, 2017.

[87] A. Mueen, S. Nath, and J. Liu. Fast approximate correlation for massive time-series data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 171–182, 2010.

[88] A. Mueen, E. J. Keogh, and N. E. Young. Logical-shapelets: an expressive primitive for time series classification. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pages 1154–1162, 2011.

[89] A. Mueen, H. Hamooni, and T. Estrada. Time series join on subsequence correlation. In *2014 IEEE International Conference on Data Mining, ICDM 2014, Shenzhen, China, December 14-17, 2014*, pages 450–459, 2014.

[90] I. Muslea. Machine learning for online query relaxation. In *KDD*, pages 246–255, 2004.

[91] I. Muslea and T. J. Lee. Online query relaxation via bayesian causal structures discovery. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 831–836, 2005.

[92] U. Nambiar and S. Kambhampati. Supporting queries with imprecise constraints. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 1654–1657, 2006.

[93] U. Nambiar and S. Kambhampati. Answering imprecise queries over autonomous web databases. In *ICDE*, page 45, 2006.

[94] A. Nandi and H. V. Jagadish. Guided interaction: Rethinking the query-result paradigm. *PVLDB*, 4(12):1466–1469, 2011.

[95] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. J. Xu. Regret-minimizing representative databases. *PVLDB*, 3(1):1114–1124, 2010.

[96] D. Nanongkai, A. Lall, A. D. Sarma, and K. Makino. Interactive regret minimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 109–120, 2012.

[97] T. Palpanas. Data series management: The road to big sequence analytics. *SIGMOD Record*, 44(2):47–52, 2015.

[98] L. Pan, J. Luo, and J. Li. Probing queries in wireless sensor networks. In *ICDCS*, pages 546–553, 2008.

[99] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *PVLDB*, 8(12):1816–1827, 2015.

[100] D. Petrov, R. Alseghayer, M. A. Sharaf, P. K. Chrysanthis, and A. Labrinidis. Interactive exploration of correlated time series. In *Proceedings of the ExploreDB'17, Chicago, IL, USA, May 19, 2017*, pages 2:1–2:6, 2017.

[101] T. Rakthanmanon, B. J. L. Campana, A. Mueen, G. E. A. P. A. Batista, M. B. Westover, Q. Zhu, J. Zakaria, and E. J. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012*, pages 262–270, 2012.

[102] G. Reeves, J. Liu, S. Nath, and F. Zhao. Managing massive time series streams with multiscale compressed trickles. *PVLDB*, 2(1):97–108, 2009.

[103] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, Nov. 2011.

[104] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1579–1590, 2014.

[105] S. Roy, L. Orr, and D. Suciu. Explaining query answers with explanation-ready databases. *PVLDB*, 9(4):348–359, 2015.

[106] S. B. Roy, H. Wang, G. Das, U. Nambiar, and M. K. Mohania. Minimum-effort driven dynamic faceted search in structured databases. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, pages 13–22, 2008.

[107] Y. Sakurai, S. Papadimitriou, and C. Faloutsos. BRAID: stream mining through group lag correlations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, pages 599–610, 2005.

[108] T. Sellam and M. L. Kersten. Cluster-driven navigation of the query space. *IEEE Trans. Knowl. Data Eng.*, 28(5):1118–1131, 2016.

[109] T. Sellam, E. Müller, and M. L. Kersten. Semi-automated exploration of data warehouses. In *Proceedings of the 24th ACM International Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015*, pages 1321–1330, 2015.

[110] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. Discovering queries based on example tuples. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 493–504, 2014.

[111] R. Shi, H. Wang, T. Wang, Y. Hou, Y. Tang, J. Li, and H. Gao. Similarity search combining query relaxation and diversification. In *Database Systems for Advanced Applications - 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part II*, pages 65–84, 2017.

[112] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Scientific discovery through weighted sampling. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 300–306, 2013.

[113] K. Stefanidis, G. Koutrika, and E. Pitoura. A survey on representation, composition and application of preferences in database systems. *ACM Trans. Database Syst.*, 36(3):19:1–19:45, 2011.

[114] B. G. Tabachnick and L. S. Fidell. *Using Multivariate Statistics (5th Edition)*. Allyn & Bacon, Inc., Needham Heights, MA, USA, 2006. ISBN 0205459382.

[115] W. C. Tan, M. Zhang, H. Elmeleegy, and D. Srivastava. Reverse engineering aggregation queries. *PVLDB*, 10(11):1394–1405, 2017.

[116] Y. Tao, X. Xiao, and J. Pei. Efficient skyline and top-k retrieval in subspaces. *IEEE Trans. Knowl. Data Eng.*, 19(8):1072–1088, 2007.

[117] A. Telang, C. Li, and S. Chakravarthy. One size does not fit all: Toward user- and query-dependent ranking for web databases. *IEEE Trans. Knowl. Data Eng.*, 24(9):1671–1685, 2012.

[118] Q. T. Tran and C.-Y. Chan. How to conquer why-not questions. In *SIGMOD Conference*, pages 15–26, 2010.

[119] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 535–548, 2009.

[120] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5): 721–746, 2014.

[121] L. H. U, M. L. Yiu, K. Mouratidis, and N. Mamoulis. Capacity constrained assignment in spatial databases. In *SIGMOD Conference*, 2008.

[122] C. Utomo, X. Li, and S. Wang. Classification based on compressive multivariate time series. In *Databases Theory and Applications - 27th Australasian Database Conference, ADC 2016, Sydney, NSW, September 28-29, 2016, Proceedings*, pages 204–214, 2016.

[123] M. Vartak, V. Raghavan, and E. A. Rundensteiner. Qrelx: generating meaningful queries that provide cardinality assurance. In *SIGMOD Conference*, pages 1215–1218, 2010.

[124] M. Vartak, S. Rahman, S. Madden, A. G. Parameswaran, and N. Polyzotis. SEEDB: efficient data-driven visualization recommendations to support visual analytics. *PVLDB*, 8 (13):2182–2193, 2015.

[125] M. Vartak, V. Raghavan, E. A. Rundensteiner, and S. Madden. Refinement driven processing of aggregation constrained queries. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016.*, pages 101–112, 2016.

[126] B. Walenz and J. Yang. Perturbation analysis of database queries. *PVLDB*, 9(14):1635–1646, 2016.

[127] A. Wasay, M. Athanassoulis, and S. Idreos. Queriosity: Automated data exploration. In *2015 IEEE International Congress on Big Data, New York City, NY, USA, June 27 - July 2, 2015*, pages 716–719, 2015.

[128] A. Wasay, X. Wei, N. Dayan, and S. Idreos. Data canopy: Accelerating exploratory statistical analysis. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 557–572, 2017.

[129] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6 (8):553–564, 2013.

[130] Z. Ye, S. Mistry, A. Bouguettaya, and H. Dong. Long-term qos-aware cloud service composition using multivariate time series analysis. *IEEE Trans. Services Computing*, 9(3): 382–393, 2016.

[131] C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, D. F. Silva, A. Mueen, and E. J. Keogh. Matrix profile I: all pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *IEEE 16th International Conference on Data Mining, ICDM 2016, December 12-15, 2016, Barcelona, Spain*, pages 1317–1322, 2016.

[132] A. Yu, P. K. Agarwal, and J. Yang. Top-k preferences in high dimensions. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 748–759, 2014.

[133] J. Zhao, F. Chevalier, E. Pietriga, and R. Balakrishnan. Exploratory analysis of time-series with chronolenses. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2422–2431, 2011.

[134] B. Zheng, N. J. Yuan, K. Zheng, X. Xie, S. W. Sadiq, and X. Zhou. Approximate keyword search in semantic trajectory database. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 975–986, 2015.

[135] K. Zoumpatianos, S. Idreos, and T. Palpanas. Indexing for interactive exploration of big data series. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1555–1566, 2014.